

Listas Ligadas

Estruturas de Dados I

Prof. Tiago A. Almeida

Departamento de Computação
Universidade Federal de São Carlos (UFSCar)

Sumário

- 1 Listas Ligadas - Discussão Intuitiva
- 2 TAD Listas e Listas Ligadas
- 3 Listas Ligadas - Implementação
- 4 Listas Ligadas com Nó Cabeça
- 5 Listas Ligadas Circulares
- 6 Listas Ligadas Ordenadas

Sumário

1 Listas Ligadas - Discussão Intuitiva

2 TAD Listas e Listas Ligadas

3 Listas Ligadas - Implementação

4 Listas Ligadas com Nó Cabeça

5 Listas Ligadas Circulares

6 Listas Ligadas Ordenadas

Discussão Intuitiva

- Ponteiros podem ser usados para construir estruturas, tais como listas, a partir de componentes simples chamados **nó**



Discussão Intuitiva

- Um **nó** possui uma seta apontando para fora. Essa seta representa um ponteiro que aponta para outro **nó**, formando uma **lista ligada**



Discussão Intuitiva

- Listas ligadas são úteis pois podem ser utilizadas para implementar o TAD lista. Nesse caso, as operações inserção e remoção no meio da lista podem ser mais eficientes
- Uma segunda vantagem é o fato de não ser necessário informar o número de elementos em tempo de compilação

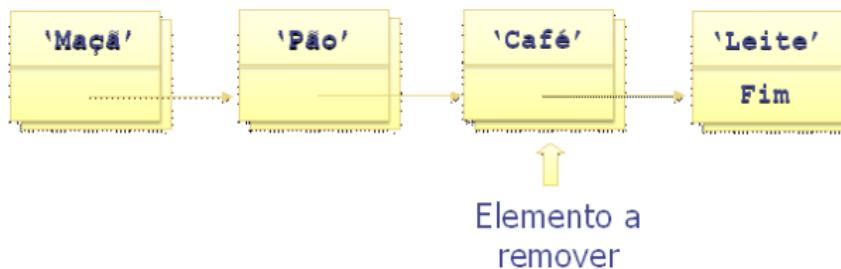
Discussão Intuitiva

- Por exemplo, uma operação de **remoção** pode ser feita da seguinte maneira



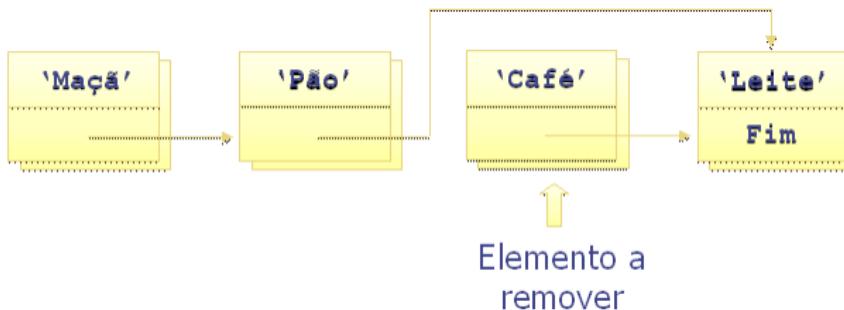
Discussão Intuitiva

- Por exemplo, uma operação de **remoção** pode ser feita da seguinte maneira



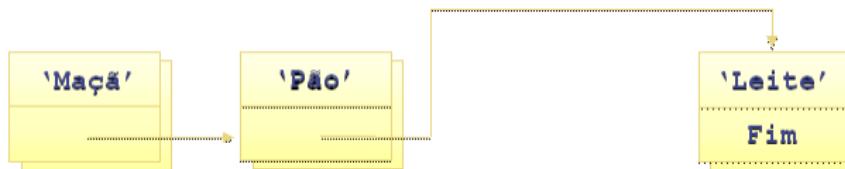
Discussão Intuitiva

- Por exemplo, uma operação de **remoção** pode ser feita da seguinte maneira



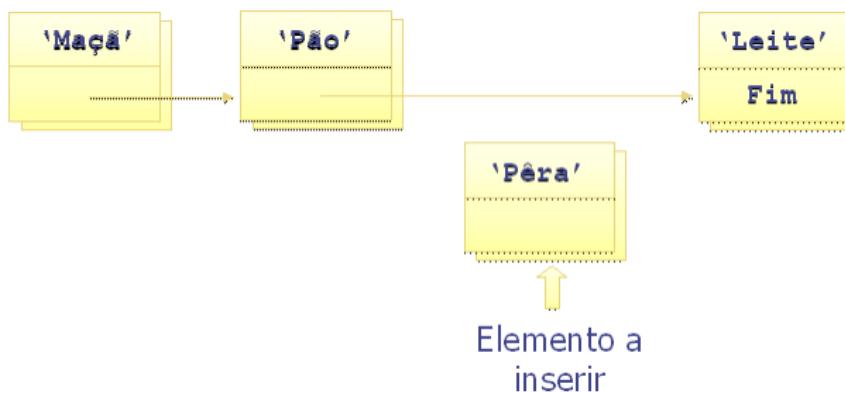
Discussão Intuitiva

- Por exemplo, uma operação de **remoção** pode ser feita da seguinte maneira



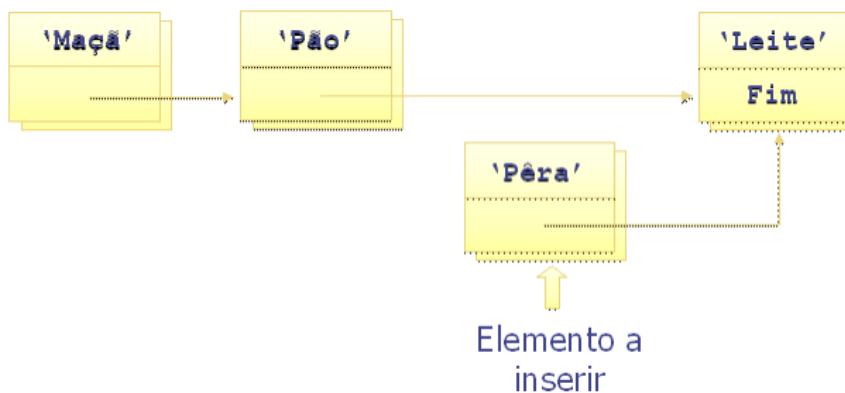
Discussão Intuitiva

- Por exemplo, uma operação de **inserção** pode ser feita da seguinte maneira



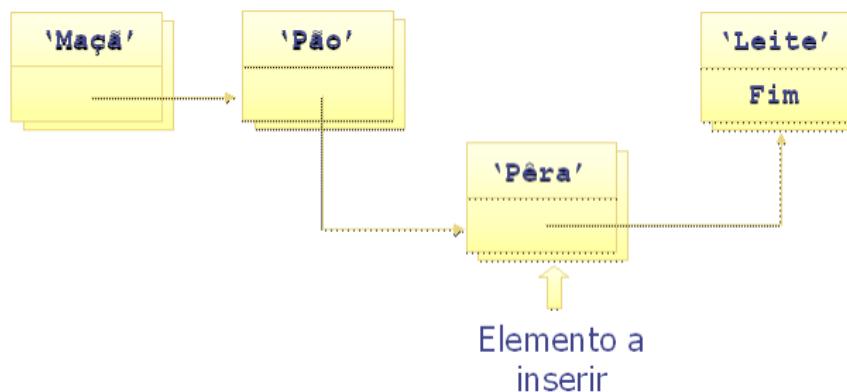
Discussão Intuitiva

- Por exemplo, uma operação de **inserção** pode ser feita da seguinte maneira



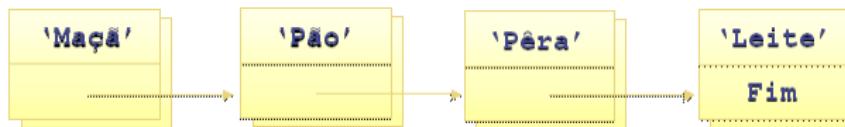
Discussão Intuitiva

- Por exemplo, uma operação de **inserção** pode ser feita da seguinte maneira



Discussão Intuitiva

- Por exemplo, uma operação de **inserção** pode ser feita da seguinte maneira



Sumário

- 1 Listas Ligadas - Discussão Intuitiva
- 2 TAD Listas e Listas Ligadas
- 3 Listas Ligadas - Implementação
- 4 Listas Ligadas com Nó Cabeça
- 5 Listas Ligadas Circulares
- 6 Listas Ligadas Ordenadas

Relembrando: TAD Listas

Principais operações

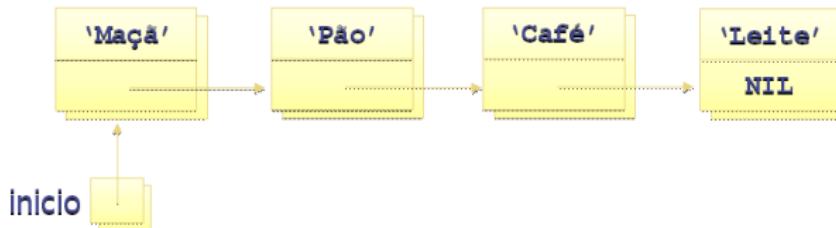
- Criar lista
- Limpar lista
- Inserir item (última posição)
- Remover item (última posição)
- Recuperar item (dado uma chave)
- Recuperar item (por posição)
- Contar número de itens
- Verificar se a lista está vazia
- Verificar se a lista está cheia
- Imprimir lista

TAD Listas e Listas Ligadas

- Antes de começarmos, precisamos definir como a lista será representada
- Uma forma bastante comum é manter uma variável ponteiro para o primeiro elemento da lista ligada

TAD Listas e Listas Ligadas

- Antes de começarmos, precisamos definir como a lista será representada
- Uma forma bastante comum é manter uma variável ponteiro para o primeiro elemento da lista ligada



TAD Listas e Listas Ligadas

- Convenciona-se que essa variável ponteiro deve ter valor **NULL** quando a lista estiver vazia
- Portanto, essa deve ser a **iniciação** da lista e também a forma de se verificar se ela se encontra vazia

TAD Listas e Listas Ligadas

- Outro detalhe importante é quanto as posições
 - Na implementação com vetores, uma posição é um valor inteiro entre 0 e o campo **fim**
 - Com listas ligadas, uma posição passa ser um ponteiro que aponta um determinado nó da lista
- Vamos analisar cada uma das operações do TAD Lista

TAD Listas I

Criar lista

- Pré-condição: nenhuma
- Pós-condição: inicia a estrutura de dados

Limpar lista

- Pré-condição: nenhuma
- Pós-condição: coloca a estrutura de dados no estado inicial

TAD Listas II

Inserir item (última posição)

- Pré-condição: **existe memória disponível**
- Pós-condição: **insere um item na última posição, retorna true se a operação foi executada com sucesso, false caso contrário**

Remover item (última posição)

- Pré-condição: a lista não está vazia
- Pós-condição: o último item da lista é removido, retorna **true** se a operação foi executada com sucesso,

TAD Listas III

Remover item (por posição)

- Pré-condição: uma posição válida da lista é informada
- Pós-condição: o item na posição fornecida é removido da lista, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

Recuperar item (dado uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

TAD Listas IV

Recuperar item (por posição)

- Pré-condição: uma posição válida da lista é informada
- Pós-condição: recupera o item na posição fornecida, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

Contar número de itens

- Pré-condição: nenhuma
- Pós-condição: retorna o número de itens na lista

TAD Listas V

Verificar se a lista está vazia

- Pré-condição: nenhuma
- Pós-condição: retorna **true** se a lista estiver vazia e **false** caso-contrário

Verificar se a lista está cheia (???)

- Pré-condição: nenhuma
- Pós-condição: retorna **true** se a lista estiver cheia e **false** caso-contrário

TAD Listas VI

Imprime lista

- Pré-condição: nenhuma
- Pós-condição: imprime na tela os itens da lista

Sumário

- 1 Listas Ligadas - Discussão Intuitiva
- 2 TAD Listas e Listas Ligadas
- 3 Listas Ligadas - Implementação
- 4 Listas Ligadas com Nó Cabeça
- 5 Listas Ligadas Circulares
- 6 Listas Ligadas Ordenadas

Lista Ligada

- Para se criar uma lista ligada, é necessário criar um **nó** que possua um ponteiro para outro **nó**

```
1 typedef struct {
2     int chave;
3     int valor;
4 } ITEM;
5
6 typedef struct NO {
7     ITEM item;
8     struct NO *proximo;
9 } NO;
```

Lista Ligada

- Considerando a estrutura NO, para a definição da lista ligada o que falta é a indicação da posição de memória do primeiro nó
- Também incluiremos a posição para o último nó para acelerar a inserção de itens no final da lista

```
1 typedef struct {  
2     NO *inicio;  
3     NO *fim;  
4 } LISTA_LIGADA;
```

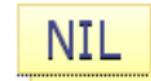
Criar lista

- Pré-condição: nenhuma
- Pós-condição: inicia a estrutura de dados

Antes

inicio 

Depois

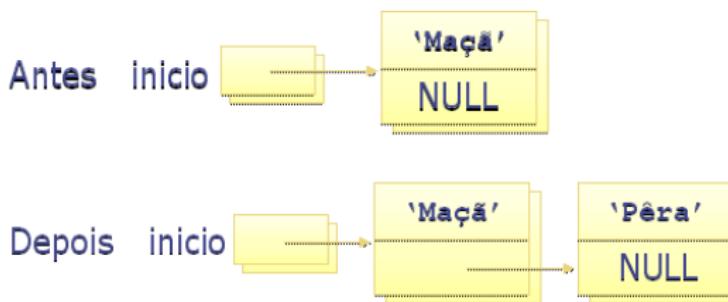
inicio 

Criar lista

```
1 void criar(struct LISTA_LIGADA *lista){  
2     lista->inicio = NULL;  
3     lista->fim = NULL;  
4 }
```

Inserir item (última posição)

- **Pré-condição:** existe memória disponível
- **Pós-condição:** insere um item na última posição, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



Memória Disponível

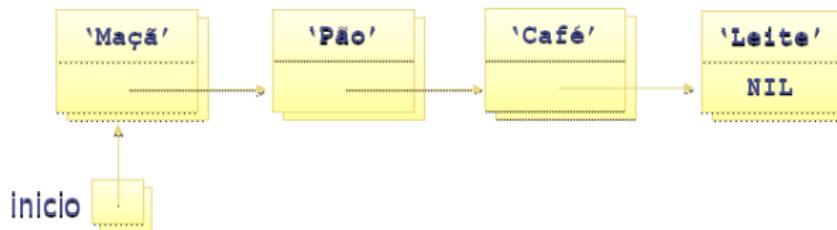
- Diferente da implementação com vetores, a lista ligada não requer especificar um tamanho para a estrutura
- Entretanto, a memória *heap* não é ilimitada e é sempre importante verificar se existe memória disponível ao chamar **malloc()**
- Em C, o procedimento **malloc()** atribui o valor **NULL** à variável ponteiro quando não existe memória disponível

Inserir item (última posição)

```
1 int inserir(LISTA_LIGADA *lista, ITEM *item) {
2     NO *pnovo = (NO *)malloc(sizeof(NO)); //cria um novo nó
3
4     if (pnovo != NULL) { //verifica se a memória foi alocada
5         pnovo->item = *item; //preenche os dados
6         pnovo->proximo = NULL; //define que o próximo é nulo
7
8         if (lista->inicio == NULL) { //se a lista for vazia
9             lista->inicio = pnovo; //inicio aponta para novo
10        } else {
11            lista->fim->proximo = pnovo; //proxímo do fim aponta para novo
12        }
13
14        lista->fim = pnovo; //fim aponta para novo
15
16        return 1;
17    } else {
18        return 0;
19    }
20 }
```

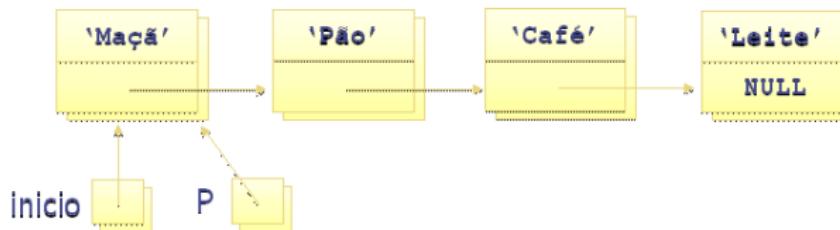
Recuperar item (dado uma chave)

- **Pré-condição:** nenhuma
- **Pós-condição:** recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



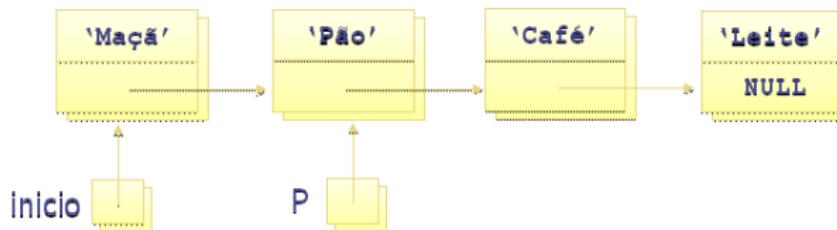
Recuperar item (dado uma chave)

- **Pré-condição:** nenhuma
- **Pós-condição:** recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



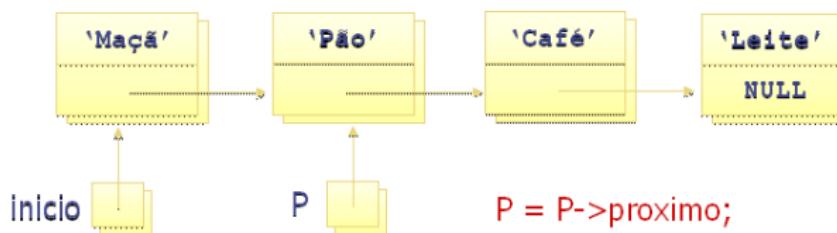
Recuperar item (dado uma chave)

- **Pré-condição:** nenhuma
- **Pós-condição:** recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



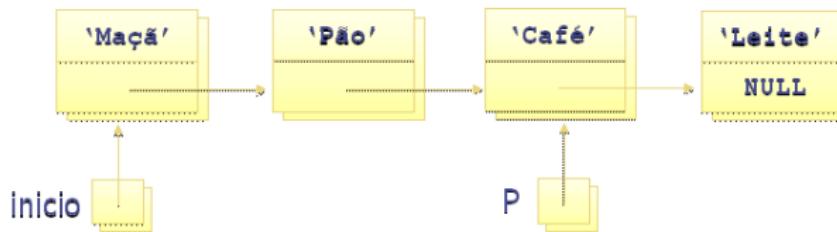
Recuperar item (dado uma chave)

- **Pré-condição:** nenhuma
- **Pós-condição:** recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



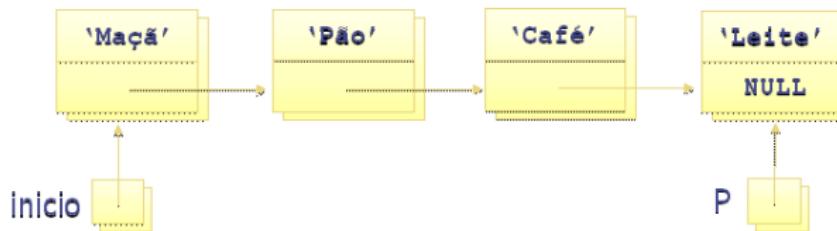
Recuperar item (dado uma chave)

- **Pré-condição:** nenhuma
- **Pós-condição:** recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



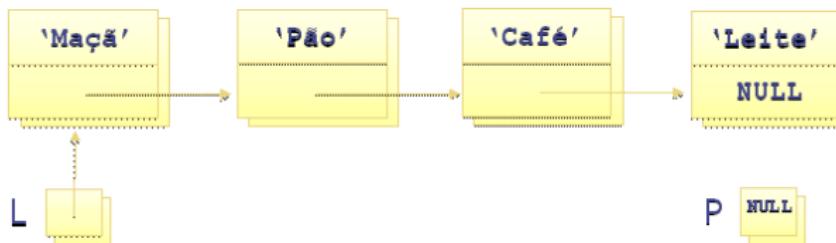
Recuperar item (dado uma chave)

- **Pré-condição:** nenhuma
- **Pós-condição:** recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



Recuperar item (dado uma chave)

- **Pré-condição:** nenhuma
- **Pós-condição:** recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



Recuperar item (dado uma chave)

```
1 int buscar(LISTA_LIGADA *lista, int chave, ITEM *item) {
2     NO *paux = lista->inicio;
3
4     while (paux != NULL) {
5         if (paux->item.chave == chave) {
6             *item = paux->item;
7             return 1;
8         }
9         paux = paux->proximo;
10    }
11
12    return 0;
13 }
```

Verificar se a lista está vazia

- **Pré-condição:** nenhuma
- **Pós-condição:** retorna **true** se a lista estiver vazia e **false** caso-contrário

```
1 int vazia(LISTA_LIGADA *lista) {  
2     return (lista->inicio == NULL);  
3 }
```

Remover item (última posição)

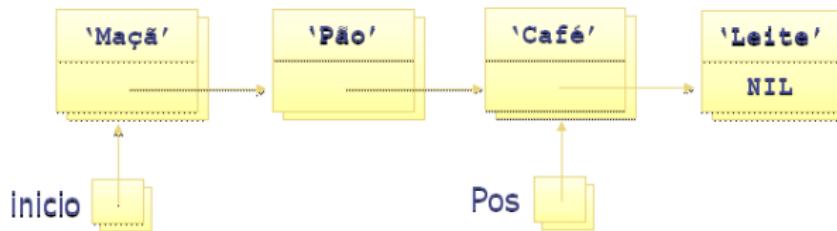
- **Pré-condição:** a lista não está vazia
- **Pós-condição:** o último item da lista é removido, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

Remover item (última posição)

```
1 int remover_fim(LISTA_LIGADA *lista) {
2     if (!vazia(lista)) {
3         //procura o penúltimo nó
4         NO *paux = lista->inicio;
5         while (paux->prox &gt;= NULL && paux->prox &gt;= lista->fim) {
6             paux = paux->prox;
7         }
8
9         if (lista->inicio == lista->fim) { //a lista tem um nó
10            free(paux->prox); //libera o único nó
11            lista->inicio = lista->fim = NULL; //lista vazia
12        } else {
13            free(lista->fim); //libera último nó
14            lista->fim = paux; //penúltimo nó vira último
15            lista->fim->prox = NULL;
16        }
17
18        return 1;
19    } else {
20        return 0;
21    }
22 }
```

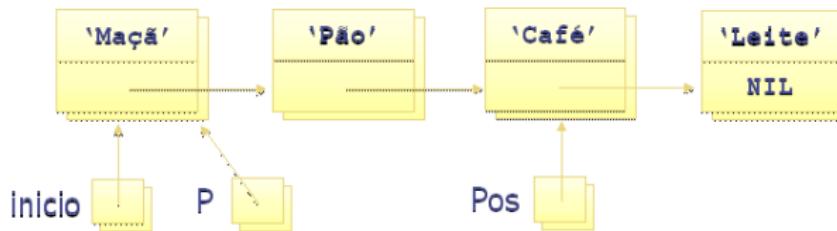
Remover item (por posição)

- **Pré-condição:** uma posição válida da lista é informada
- **Pós-condição:** o item na posição fornecida é removido da lista, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



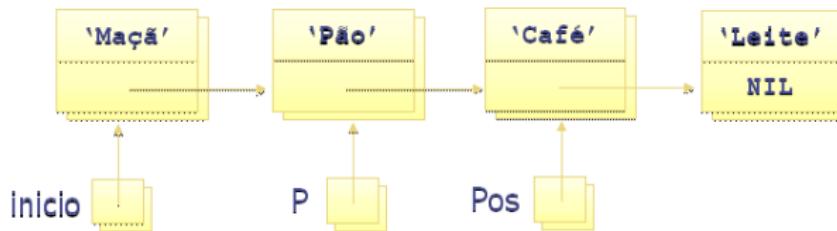
Remover item (por posição)

- **Pré-condição:** uma posição válida da lista é informada
- **Pós-condição:** o item na posição fornecida é removido da lista, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



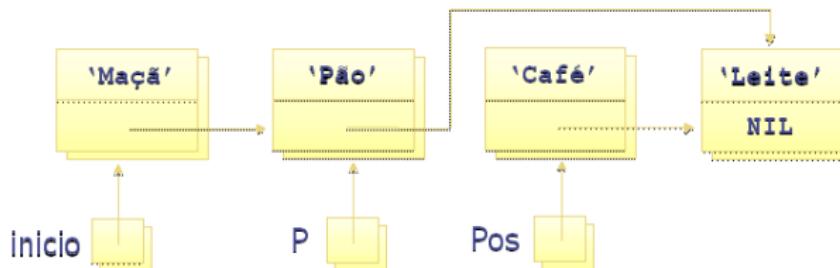
Remover item (por posição)

- **Pré-condição:** uma posição válida da lista é informada
- **Pós-condição:** o item na posição fornecida é removido da lista, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



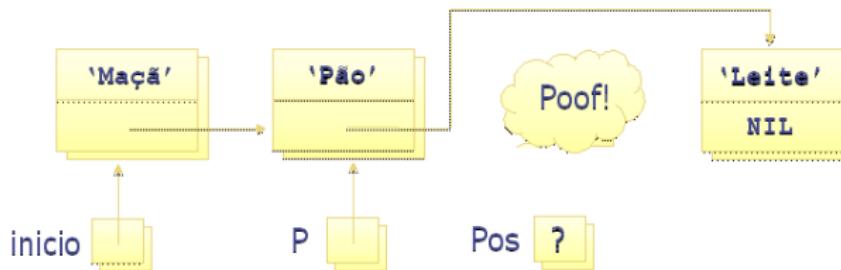
Remover item (por posição)

- **Pré-condição:** uma posição válida da lista é informada
- **Pós-condição:** o item na posição fornecida é removido da lista, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



Remover item (por posição)

- **Pré-condição:** uma posição válida da lista é informada
- **Pós-condição:** o item na posição fornecida é removido da lista, retorna **true** se a operação foi executada com sucesso, **false** caso contrário



Exercício

Implementar as demais operações do TAD Listas

- Limpar lista
- Inserir item (por posição)
- Remover item (por posição)
- Recuperar item (por posição)
- Contar número de itens
- Imprimir lista

Exercício

Exercícios

Crie funções que implementem as seguintes operações:

- Verificar se a lista **L** está ordenada (crescente ou decrescente)
- Fazer uma cópia da Lista **L1** em outra **L2**
- Fazer uma cópia da Lista **L1** em **L2**, eliminando repetidos
- Inverter **L1**, colocando o resultado em **L2**
- Inverter a própria **L1**
- Intercalar **L1** com **L2**, gerando **L3** ordenada (considere **L1** e **L2** ordenadas)
- Eliminar de **L1** todas as ocorrências de um dado item (**L1** está ordenada)

Sumário

- 1 Listas Ligadas - Discussão Intuitiva
- 2 TAD Listas e Listas Ligadas
- 3 Listas Ligadas - Implementação
- 4 Listas Ligadas com Nó Cabeça
- 5 Listas Ligadas Circulares
- 6 Listas Ligadas Ordenadas

Introdução

- Das operações anteriores, a mais **complexa** é a **remoção** de um elemento do meio da lista
- Isso porque o algoritmo precisa **apontar para o item anterior ao que será removido**, o que no caso da **remoção do primeiro elemento** se configura uma **exceção** que precisa ser tratada a parte
- Uma solução que simplifica a implementação é **substituir** o ponteiro para **início** por um **nó cabeça**
- Um **nó cabeça** é um nó normal da lista, mas esse é **sempre o primeiro nó** e a **informação** armazenada **não tem valor**

Nó Cabeça e Lista Vazia

- A lista com nó cabeça será vazia quando o **próximo** do nó cabeça apontar para **NULL**

```
1 typedef struct {  
2     NO cabeca;  
3     NO *fim;  
4 } LISTA_LIGADA;  
5  
6 int vazia(LISTA_LIGADA *lista) {  
7     return (lista->cabeca.proximo == NULL);  
8 }
```

Implementação das Demais Operações

- A implementação das demais operações é similar a lista ligada padrão (sem nó cabeça), a única alteração é substituir as referências ao ponteiro `início` pelo próximo do nó cabeça
- O grande ganho é na remoção do meio da lista (por posição), já que nesse não é necessário tratar separadamente quando o item a se remover é o primeiro

Remover item (por posição)

```
1 int remover_posicao(LISTA_LIGADA *lista, int pos) {
2     int i;
3
4     if (!vazia(lista)) { //verifico se a lista está vazia
5         NO *paux = &lista->cabeca; //aponta para o elemento anterior a ser ←
6             retirado
7
8         for (i = 0; i < pos; i++) {
9             if (paux->proximo != lista->fim) {
10                 paux = paux->proximo;
11             } else {
12                 return 0;
13             }
14
15         NO *prem = paux->proximo;
16         paux->proximo = paux->proximo->proximo;
17
18         if (prem->proximo == NULL) { //retirei o último item
19             lista->fim = paux;
20         }
21
22         free(prem);
23
24         return 1;
25     } else {
26         return 0;
27     }
28 }
```

Exercício

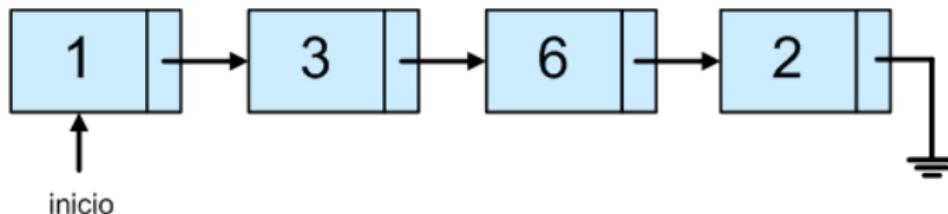
- Implementar as demais operações do TAD listas usando o conceito de lista ligada com nó cabeça

Sumário

- 1 Listas Ligadas - Discussão Intuitiva
- 2 TAD Listas e Listas Ligadas
- 3 Listas Ligadas - Implementação
- 4 Listas Ligadas com Nó Cabeça
- 5 Listas Ligadas Circulares
- 6 Listas Ligadas Ordenadas

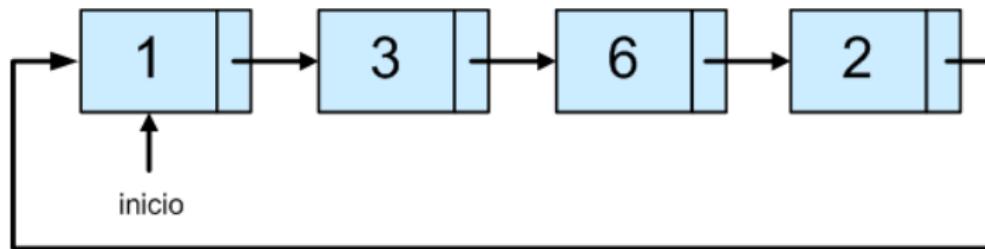
Listas Ligadas Circulares

- Um diferente tipo de implementação de listas ligadas substitui a definição de que o próximo do último é **NULL** por a próximo do último é o primeiro



Listas Ligadas Circulares

- Um diferente tipo de implementação de listas ligadas substitui a definição de que o próximo do último é **NULL** por a próximo do último é o primeiro

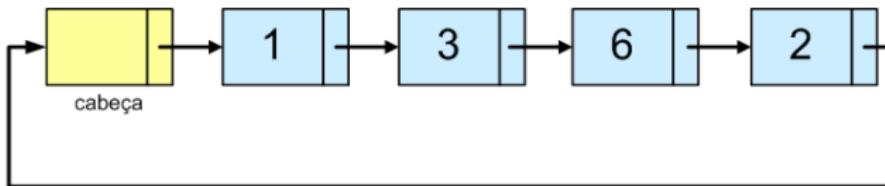


Listas Ligadas Circulares

- A partir de um nó da lista pode-se chegar a qualquer outro nó
- Nessa implementação somente um ponteiro para o fim da lista é necessário, não sendo necessário um ponteiro para o início. Isso porque o **início** é o próximo do fim

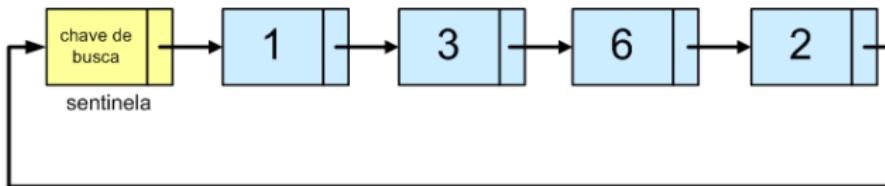
Listas Ligadas Circulares (Sentinela)

- No caso especial da busca em listas circulares, o emprego de um nó cabeça pode reduzir a quantidade de testes necessários
- A ideia é colocar a chave de busca no nó cabeça e começar a busca no próximo nó
- Se o item encontrado for a cabeça, a busca não teve sucesso. Assim um teste é “economizado” já que não é preciso testar se a lista acabou
- Nesse caso, o nó cabeça é chamado de **sentinela**



Listas Ligadas Circulares (Sentinela)

- No caso especial da busca em listas circulares, o emprego de um nó cabeça pode reduzir a quantidade de testes necessários
- A ideia é colocar a chave de busca no nó cabeça e começar a busca no próximo nó
- Se o item encontrado for a cabeça, a busca não teve sucesso. Assim um teste é “economizado” já que não é preciso testar se a lista acabou
- Nesse caso, o nó cabeça é chamado de **sentinela**



Listas Ligadas Circulares (Sentinela)

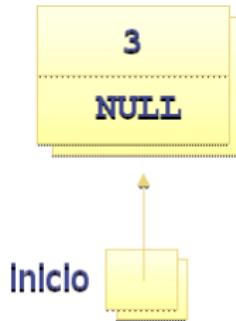
```
1 typedef struct {
2     NO sentinel;
3     NO *fim;
4 } LISTA_LIGADA_CIRCULAR;
5
6
7 int buscar(LISTA_LIGADA_CIRCULAR *lista, int chave, ITEM *item) {
8     lista->sentinel.item.chave = chave; //atribui a chave ao sentinel
9
10    NO *paux = &lista->sentinel;
11
12    do {
13        paux = paux->proximo;
14    } while (paux->item.chave != chave);
15
16    *item = paux->item; //retorno o valor encontrado
17
18    return (paux != &lista->sentinel); //verifico se o valor não é a ←
19    sentinel
}
```

Sumário

- 1 Listas Ligadas - Discussão Intuitiva
- 2 TAD Listas e Listas Ligadas
- 3 Listas Ligadas - Implementação
- 4 Listas Ligadas com Nó Cabeça
- 5 Listas Ligadas Circulares
- 6 Listas Ligadas Ordenadas

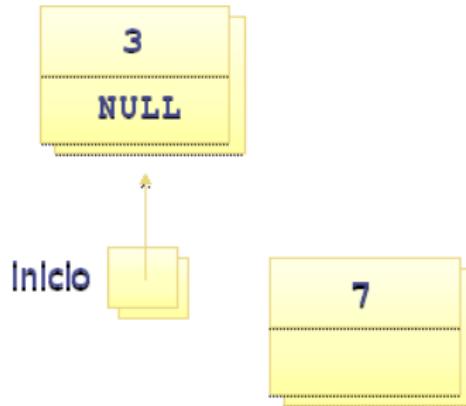
Inserção Ordenada - Lista Ligada

- Inserindo valor 3



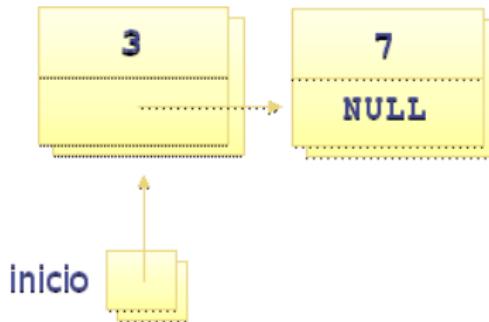
Inserção Ordenada - Lista Ligada

- Inserindo valor 7



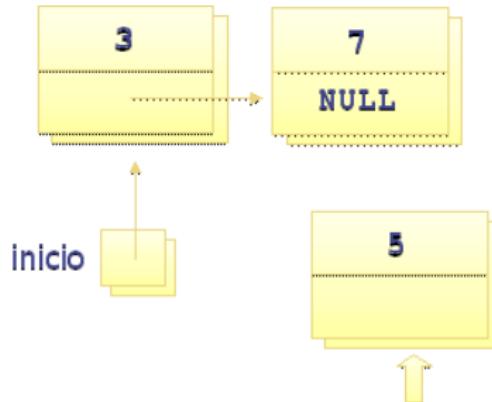
Inserção Ordenada - Lista Ligada

- Inserindo valor 7



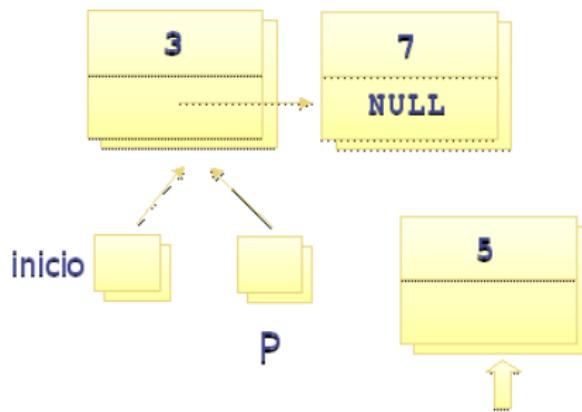
Inserção Ordenada - Lista Ligada

- Inserindo valor 5



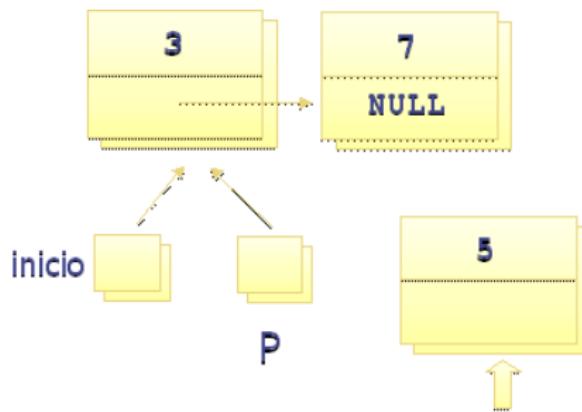
Inserção Ordenada - Lista Ligada

- Inserindo valor 5
- inicio.chave menor que novo.chave



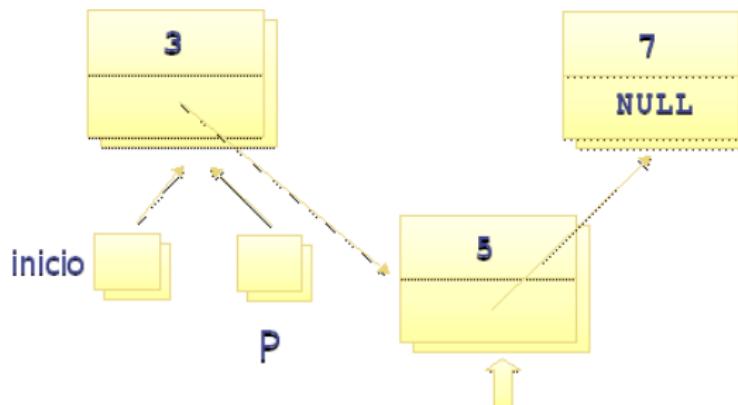
Inserção Ordenada - Lista Ligada

- Inserindo valor 5
- $p \rightarrow \text{proximo.chave}$ maior que novo.chave



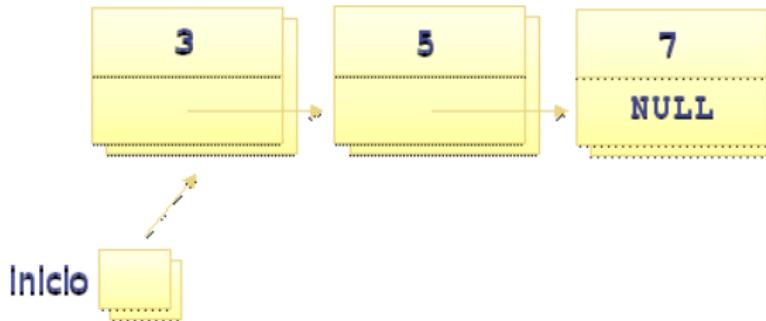
Inserção Ordenada - Lista Ligada

- Inserindo valor 5



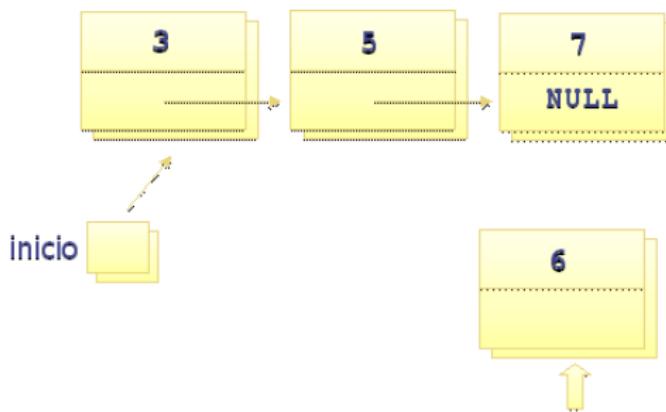
Inserção Ordenada - Lista Ligada

- Inserindo valor 5



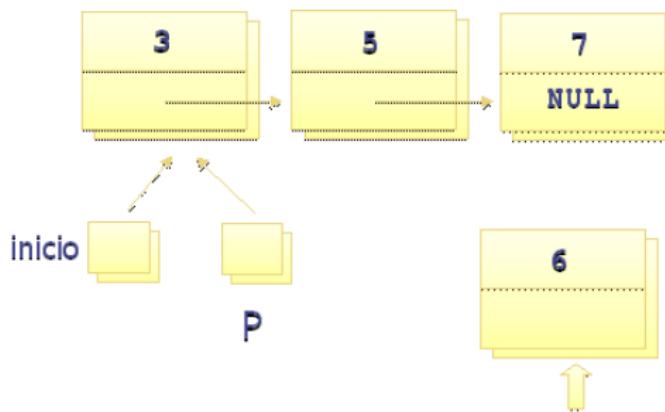
Inserção Ordenada - Lista Ligada

- Inserindo valor 6
- inicio.chave menor que novo.chave



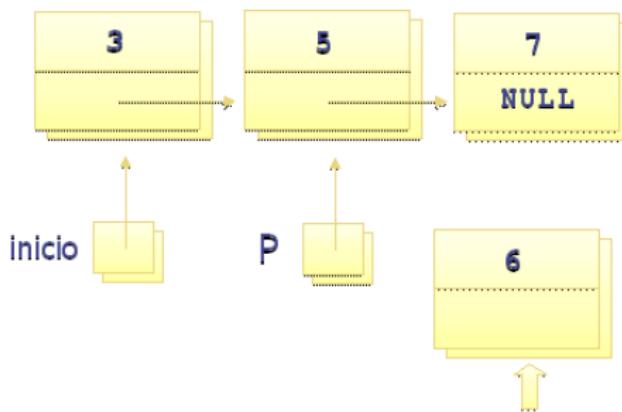
Inserção Ordenada - Lista Ligada

- Inserindo valor 6
- $p \rightarrow \text{proximo.chave}$ menor que novo.chave



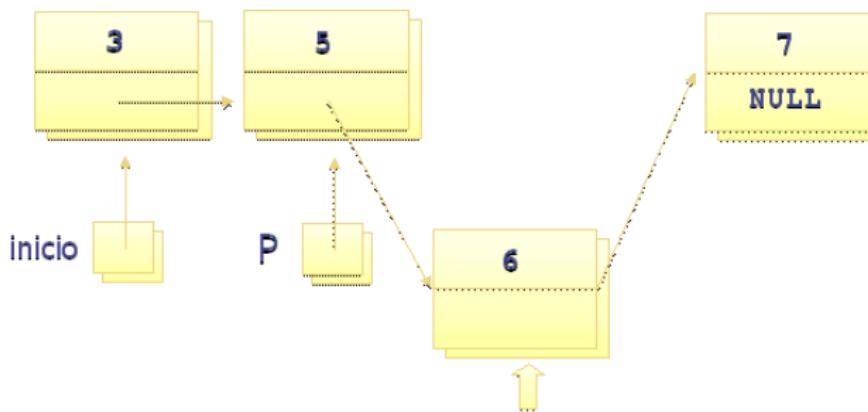
Inserção Ordenada - Lista Ligada

- Inserindo valor 6
- $p \rightarrow \text{proximo.chave}$ maior que novo.chave



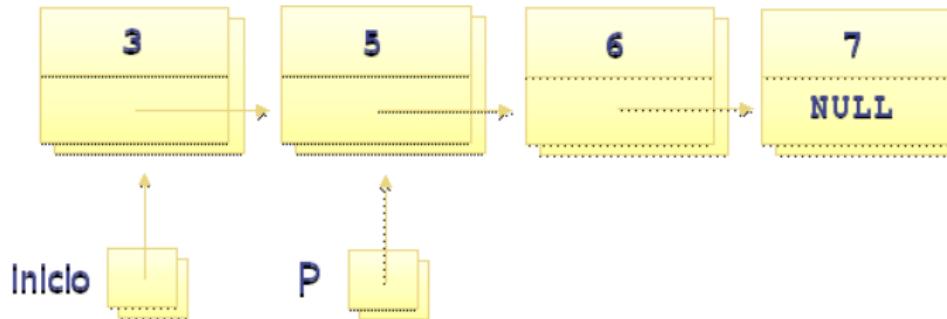
Inserção Ordenada - Lista Ligada

- Inserindo valor 6
- $p \rightarrow \text{proximo.chave}$ maior que novo.chave



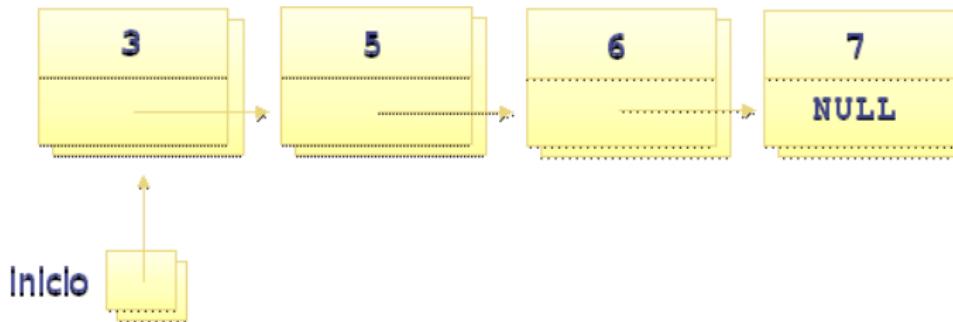
Inserção Ordenada - Lista Ligada

- Inserindo valor 6



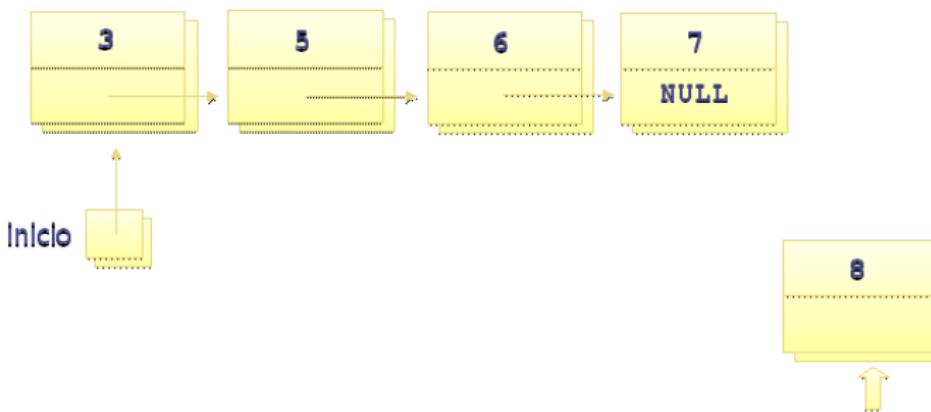
Inserção Ordenada - Lista Ligada

- Inserindo valor 6



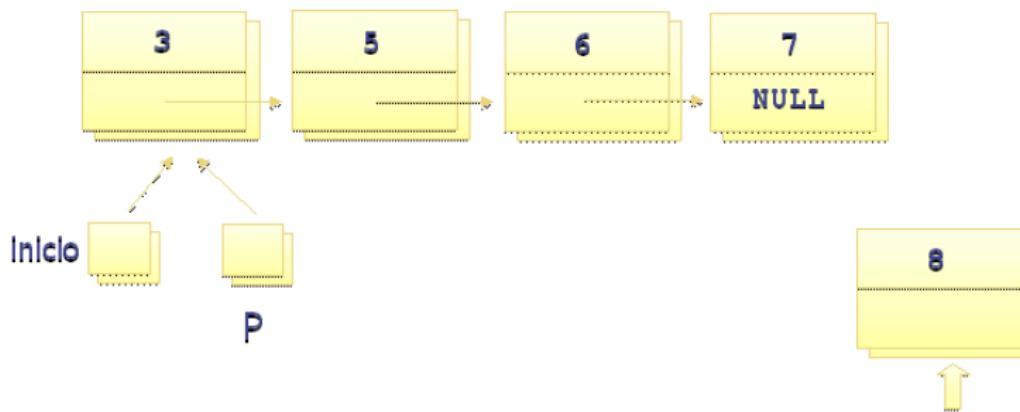
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- inicio.chave **menor** que novo.chave



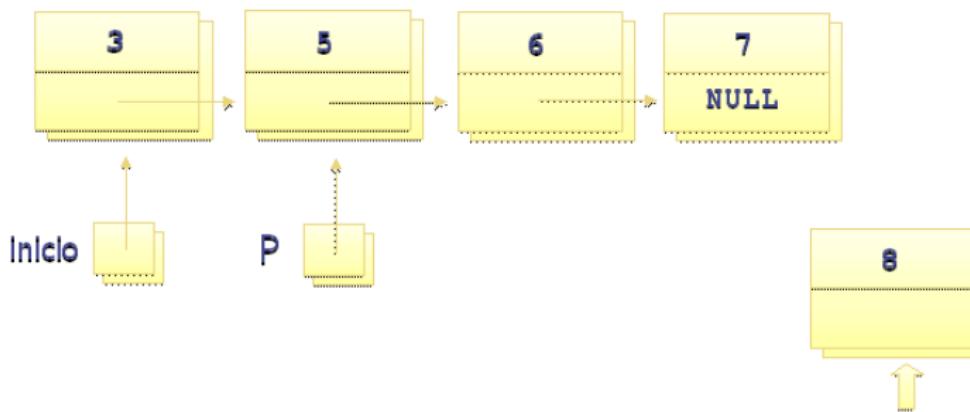
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- $p \rightarrow \text{proximo.chave}$ menor que novo.chave



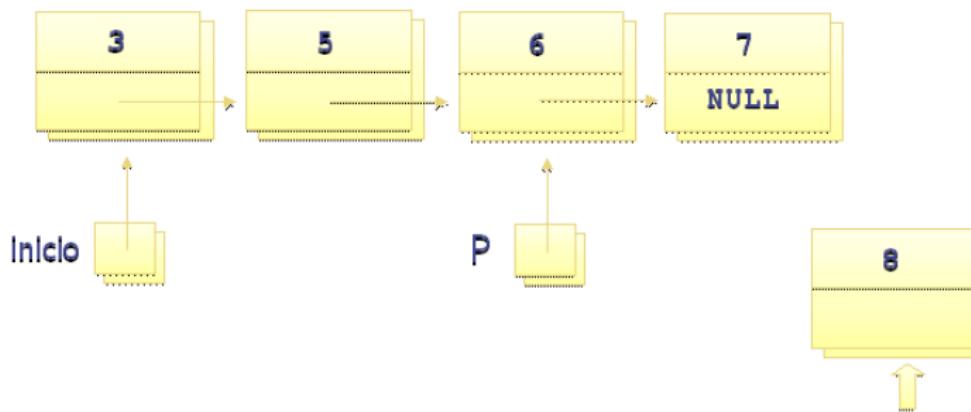
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- $p \rightarrow \text{proximo.chave}$ menor que novo.chave



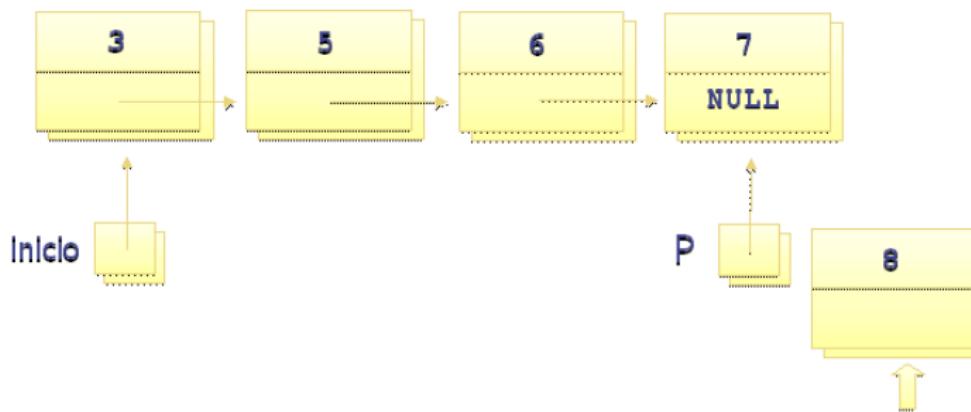
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- $p \rightarrow \text{proximo.chave}$ menor que novo.chave



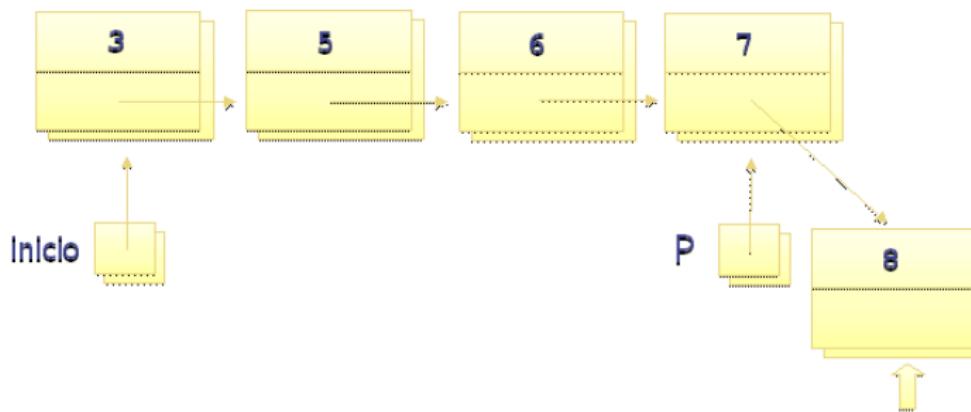
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- P->proxímo não existe



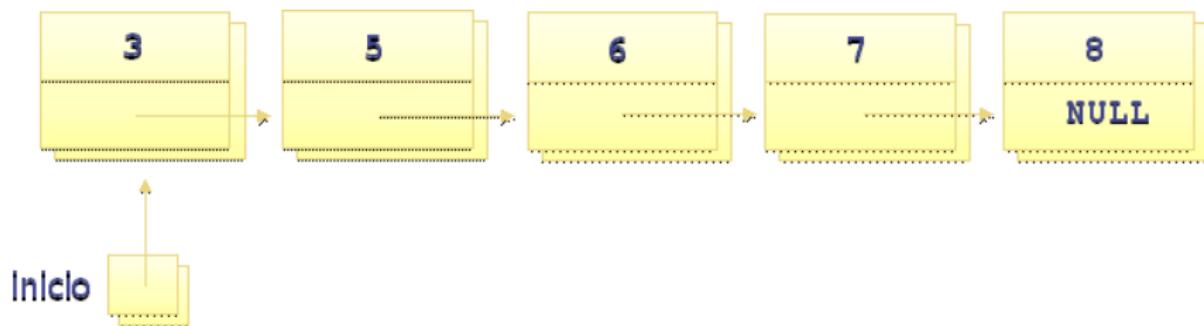
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- P->proxímo não existe



Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- P->proxímo **não existe**



Comentários sobre a Implementação

- Não precisa do ponteiro fim porque a inserção será em qualquer posição de lista
- Novamente o emprego do nó cabeça facilita a implementação uma vez que vamos buscar a posição anterior da de inserção, e no caso de ser o menor item da lista isso não representará exceção

Inserção Ordenada - Implementação

```
1 int inserir(LISTA_LIGADA *lista, ITEM *item) {
2     NO *pnovo = (NO *)malloc(sizeof(NO)); //crio um novo nó
3
4     if (pnovo != NULL) {
5         pnovo->item = *item; //preencho os dados
6         pnovo->proximo = NULL; //defino que o próximo é nulo
7
8         NO *paux = &lista->cabeca; //armazena posição anterior da de inserção
9
10        while ((paux->proximo != NULL) &&
11              (paux->proximo->item.chave < item->chave)) {
12            paux = paux->proximo;
13        }
14
15        pnovo->proximo = paux->proximo;
16        paux->proximo = pnovo;
17
18        return 1;
19    } else {
20        return 0;
21    }
22 }
```

Busca em Lista Ordenada

- Lembrete: é possível tirar vantagem em uma busca se a lista é ordenada

```
1 int buscar(LISTA_LIGADA *lista, int chave, ITEM *item) {
2     NO *paux = lista->cabeca.proximo;
3
4     while (paux != NULL) { //precorre a lista
5         if (paux->item.chave == chave) { //se a chave for igual
6             *item = paux->item;
7             return 1;
8         } else if (paux->item.chave > chave) { //se a chave na lista for ←
9             maior
10            return 0;
11        }
12
13        paux = paux->proximo;
14    }
15
16    return 0;
}
```

Lista Ordenada - Outras Operações

- As demais operações implementadas podem deixar a lista desordenada?
- Poderia ocorrer com a remoção de elementos, entretanto
 - Com vetores, a implementação deslocava os elementos
 - Com listas ligadas, os nós removidos não alteram a ordem dos demais