Listas Ligadas

SCC0202 - Algoritmos e Estruturas de Dados I

Prof. Fernando V. Paulovich
**Baseado no material do Prof. Gustavo Batista*
http://www.icmc.usp.br/~paulovic
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

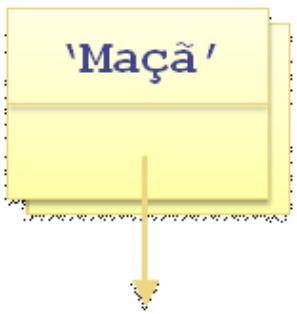
4 de setembro de 2013

Sumário

Sumário

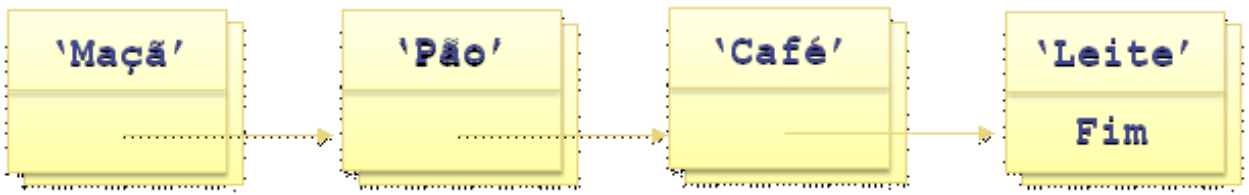
Discussão Intuitiva

- 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

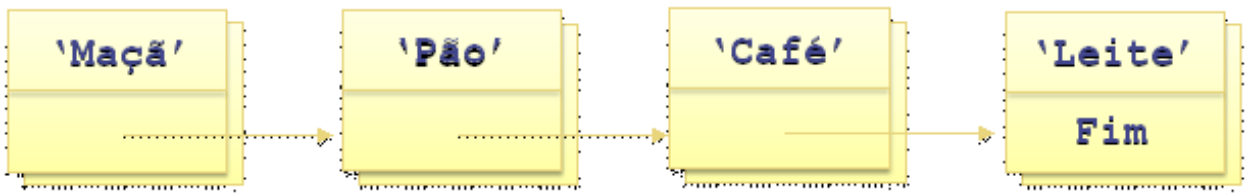
- Listas ligadas são úteis pois podem ser utilizadas para implementar o TAD lista. Nesse caso, as operações inserção (ordenada) e remoção no meio da lista podem ser mais eficientes

Discussão Intuitiva

- Listas ligadas são úteis pois podem ser utilizadas para implementar o TAD lista. Nesse caso, as operações inserção (ordenada) 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

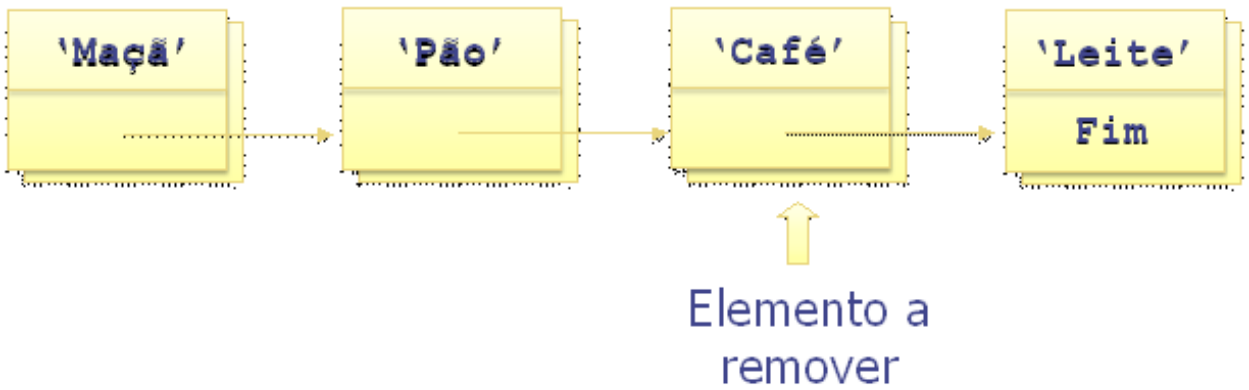
Discussão Intuitiva

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



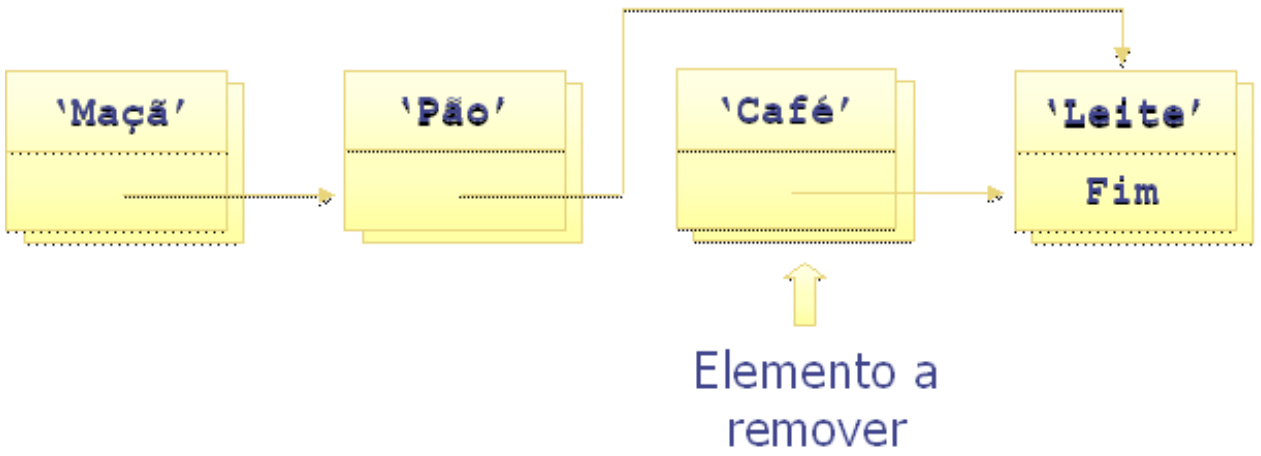
Discussão Intuitiva

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



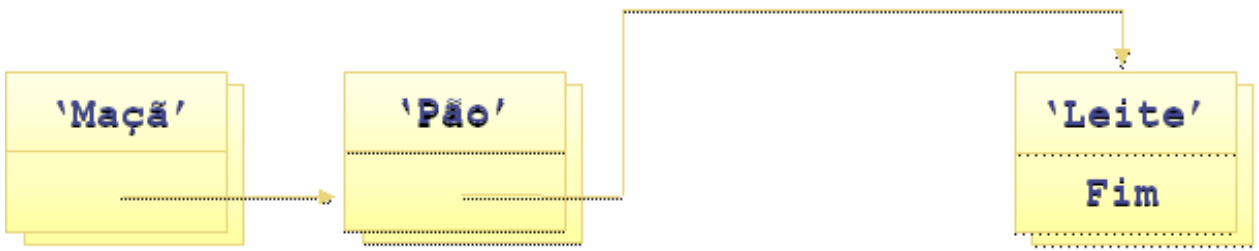
Discussão Intuitiva

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



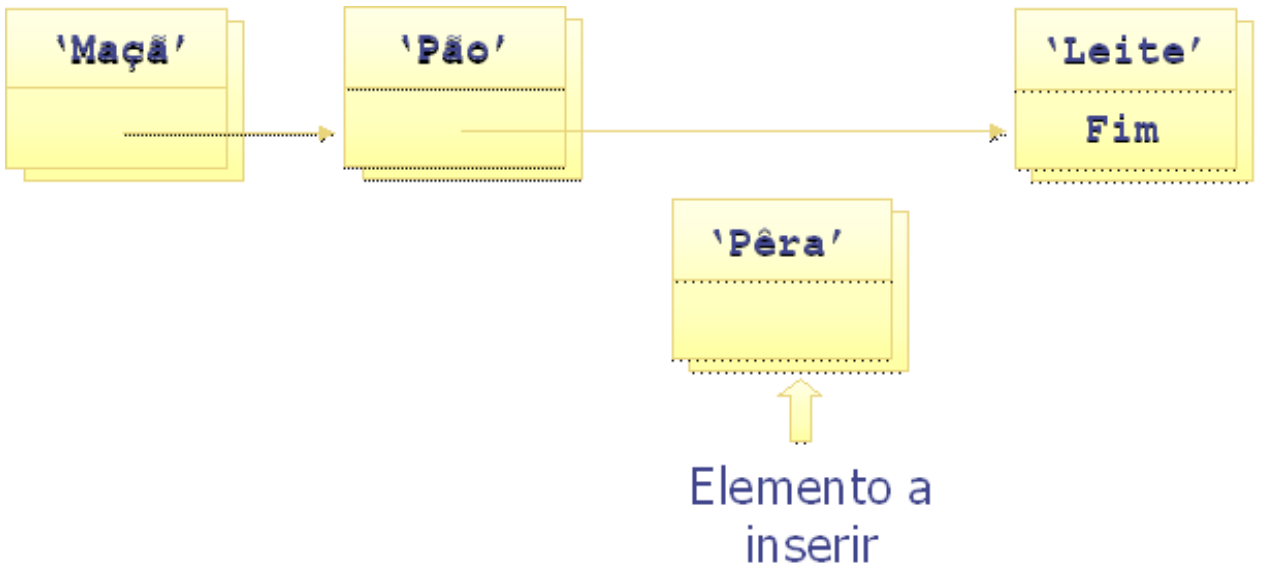
Discussão Intuitiva

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



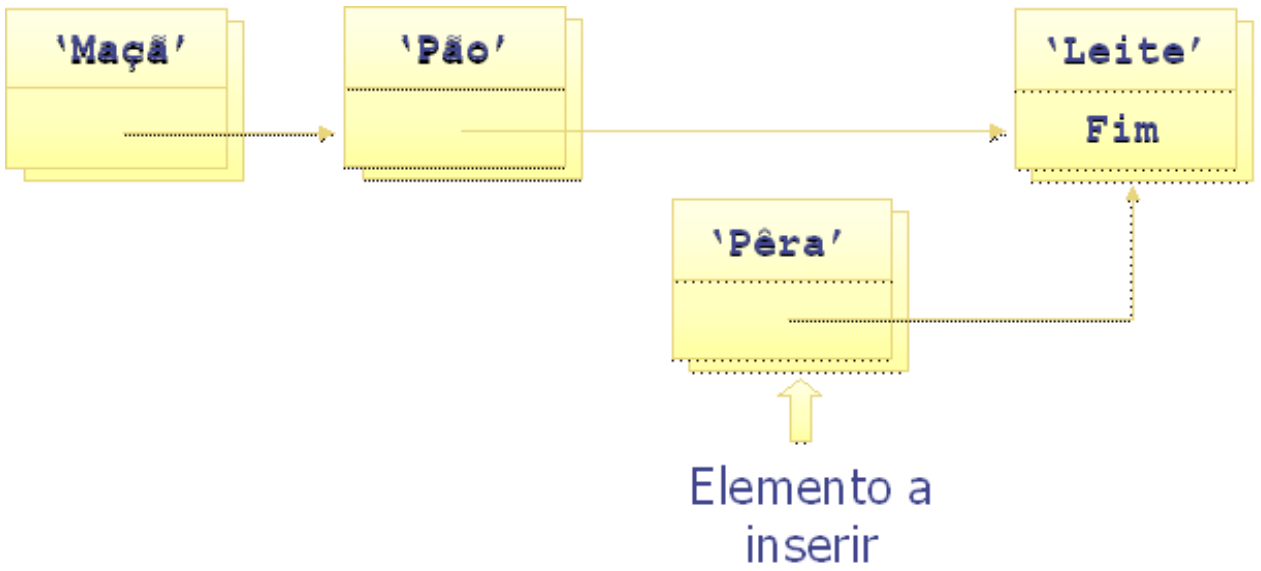
Discussão Intuitiva

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



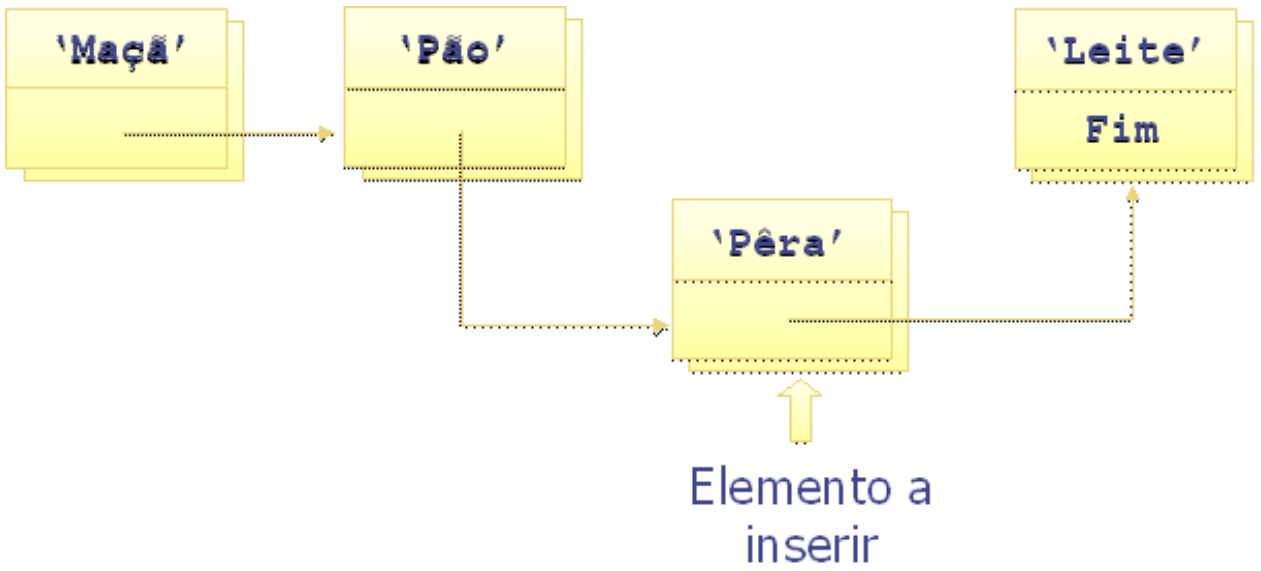
Discussão Intuitiva

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



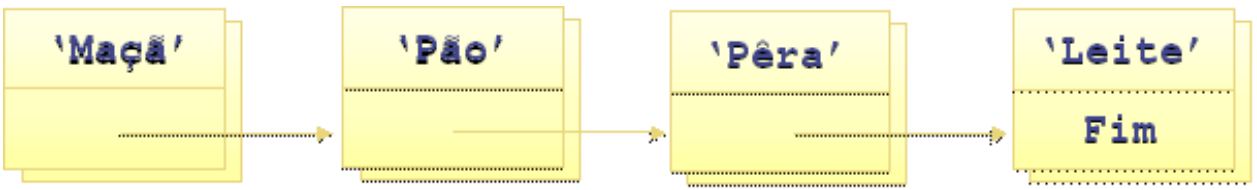
Discussão Intuitiva

- 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

Relembrando: TAD Listas

Principais operações

- Criar lista
- Apagar lista
- Inserir item (última posição)
- Remover item (dado uma chave)
- Recuperar item (dado uma chave)
- 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

TAD Listas e Listas Ligadas

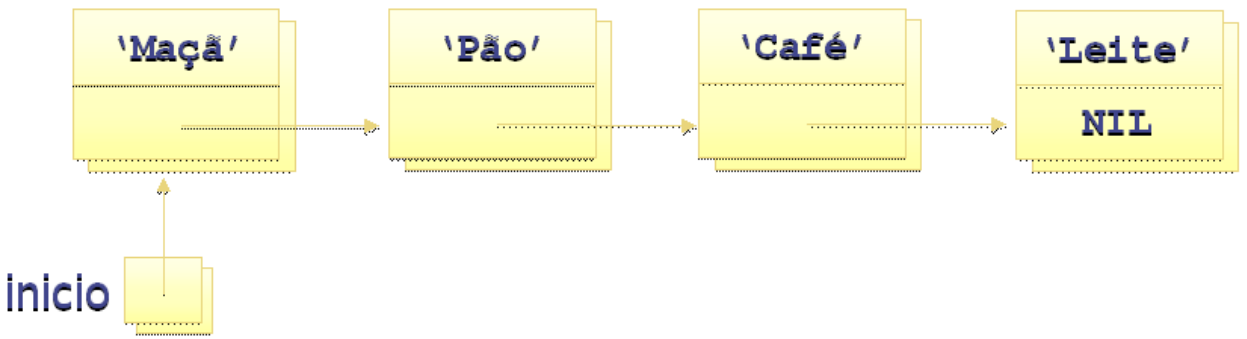
- Antes de começarmos, precisamos definir como a lista será representada
- Uma forma bastante comum é manter uma variável ponteiro para o primeiro elemento da lista ligada

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

- 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

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

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**

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

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: existir espaço na memória
- Pós-condição: inicia a estrutura de dados

Limpar lista

- Pré-condição: nenhuma
- Pós-condição: remove a estrutura de dados da memória

TAD Listas II

Inserir item

- 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 (dado uma chave)

- Pré-condição: nenhuma
- Pós-condição: remove um determinado item da lista dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

TAD Listas III

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

Contar número de itens

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

TAD Listas IV

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 V

Imprimir lista

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

Sumário

Listas Ligadas

```
1  #ifndef LISTALIGADA_H
2  #define LISTALIGADA_H
3
4  #include "Item.h"
5
6  typedef struct lista_ligada LISTA_LIGADA;
7
8  LISTA_LIGADA *criar_lista();
9  void apagar_lista(LISTA_LIGADA **lista);
10
11 int inserir_item(LISTA_LIGADA *lista, ITEM *item);
12 int remover_item(LISTA_LIGADA *lista, int chave);
13 ITEM *recuperar_item(LISTA_LIGADA *lista, int chave);
14
15 int vazia(LISTA_LIGADA *lista);
16 int cheia(LISTA_LIGADA *lista);
17 int tamanho(LISTA_LIGADA *lista);
18 void imprimir(LISTA_LIGADA *lista);
19
20 #endif
```

Lista Ligada

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

Lista Ligada

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

```
1 typedef struct NO {
2   ITEM *item;
3   struct NO *proximo;
4 } NO;
```

Função para Apagar Nó

```
1 void apagar_no(NO *no) {
2   apagar_item(&(no->item));
3   free(no);
4 }
```

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

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 e uma variável tamanho

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 e uma variável tamanho

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 e uma variável tamanho

```
1 struct lista_ligada {
2   NO *inicio;
3   NO *fim;
4   int tamanho;
5 };
```

Criar lista

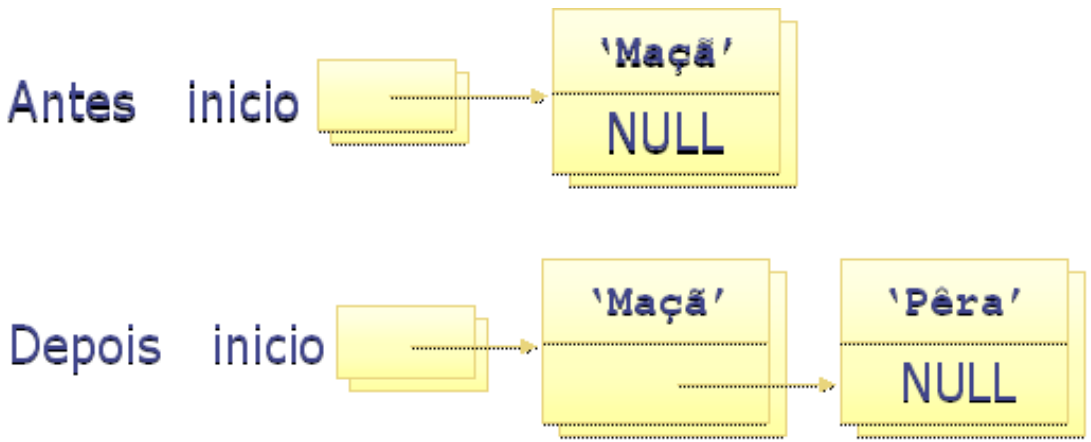
- **Pré-condição**: nenhuma
- **Pós-condição**: inicia a estrutura de dados

Criar lista

```
1  LISTA_LIGADA *criar_lista() {
2    LISTA_LIGADA *lista = (LISTA_LIGADA *)malloc(sizeof(↩
         LISTA_LIGADA));
3
4    if(lista != NULL) {
5      lista->inicio = NULL;
6      lista->fim = NULL;
7      lista->tamanho = 0;
8    }
9
10   return lista;
11 }
```

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

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()**

Memória Disponível

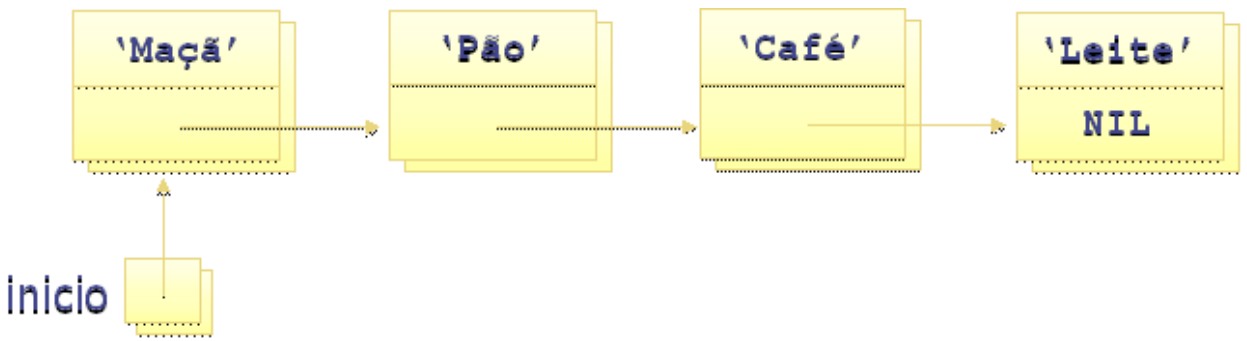
- Diferente da implementação com vetores, a lista ligada não requer especificar um tamanho para a estrutura
- Entretanto, a memória *heap* não é ilimitada e é sempre importante verificar se existe memória disponível ao chamar **malloc()**
- Em C, o procedimento **malloc()** atribui o valor **NULL** à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

Inserir item (última posição)

```
1  int inserir_item(LISTA_LIGADA *lista, ITEM *item) {
2    NO *pnovo = (NO *) malloc(sizeof (NO));
3
4    if (pnovo != NULL) {
5      pnovo->item = item;
6      pnovo->proximo = NULL;
7
8      if (lista->inicio == NULL) {
9        lista->inicio = pnovo;
10     } else {
11       lista->fim->proximo = pnovo;
12     }
13
14     lista->fim = pnovo;
15     lista->tamanho++;
16     return 1;
17   } else {
18     return 0;
19   }
20 }
```

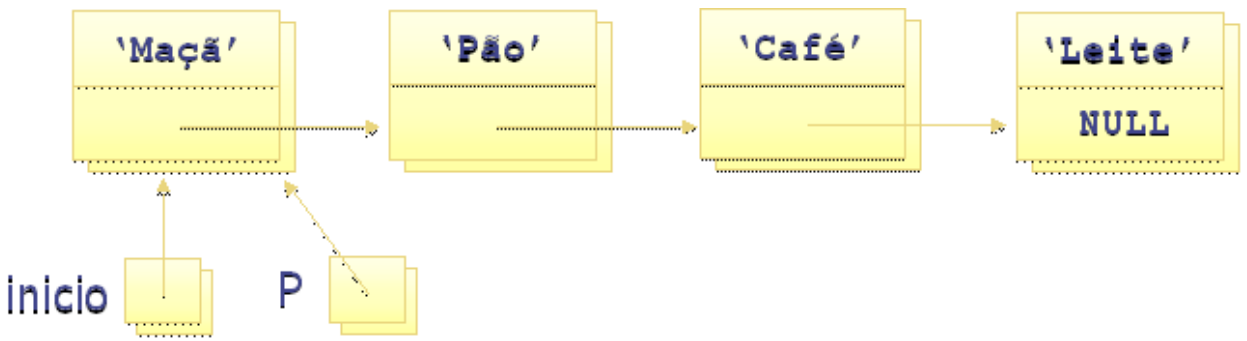
Recuperar item (dado uma chave)

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



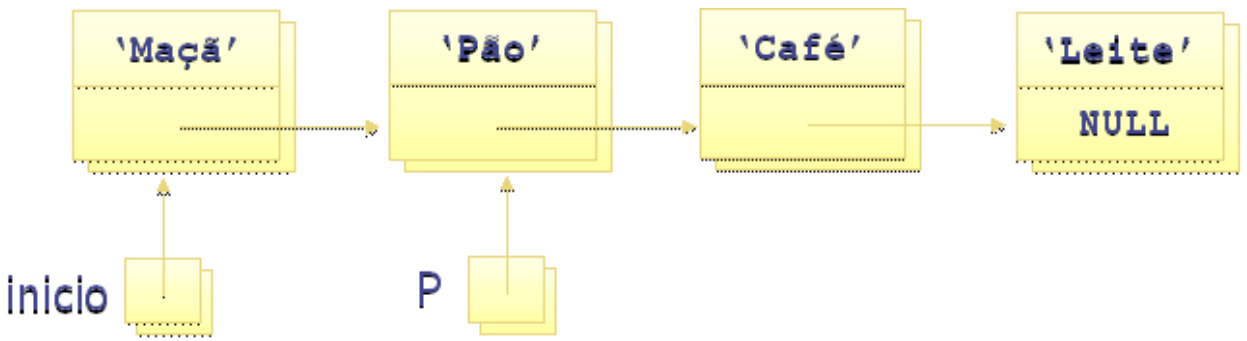
Recuperar item (dado uma chave)

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



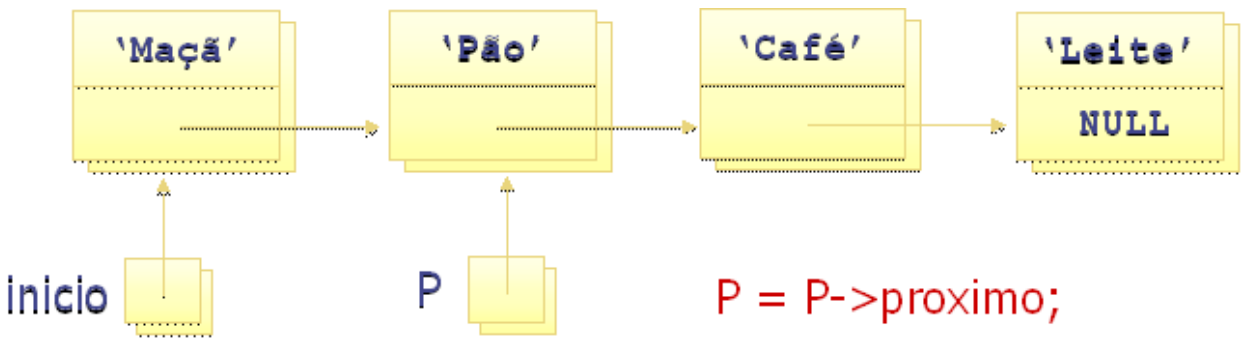
Recuperar item (dado uma chave)

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



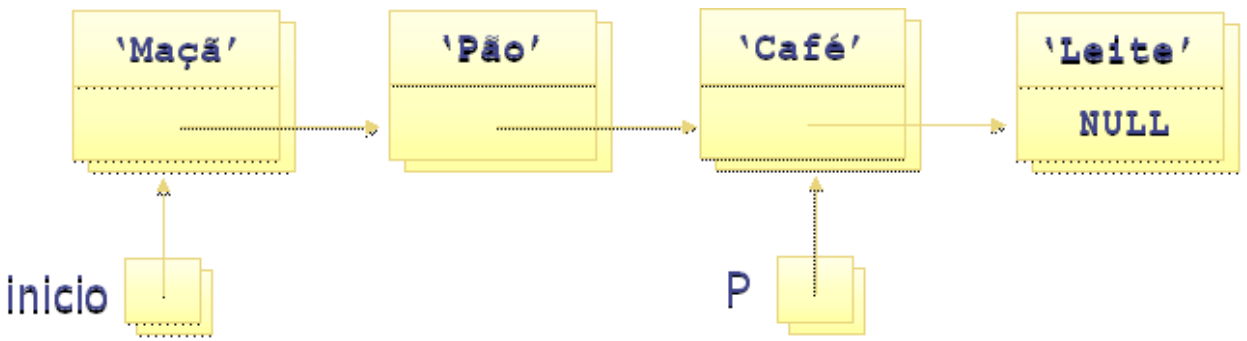
Recuperar item (dado uma chave)

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



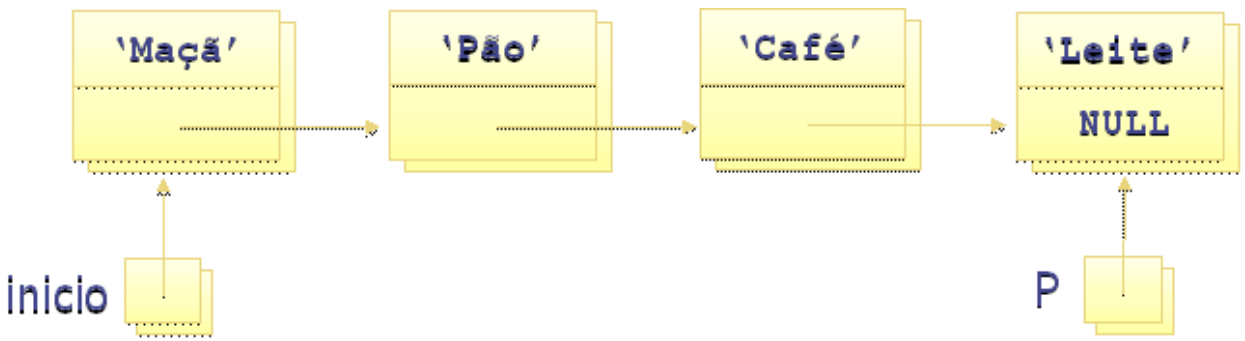
Recuperar item (dado uma chave)

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



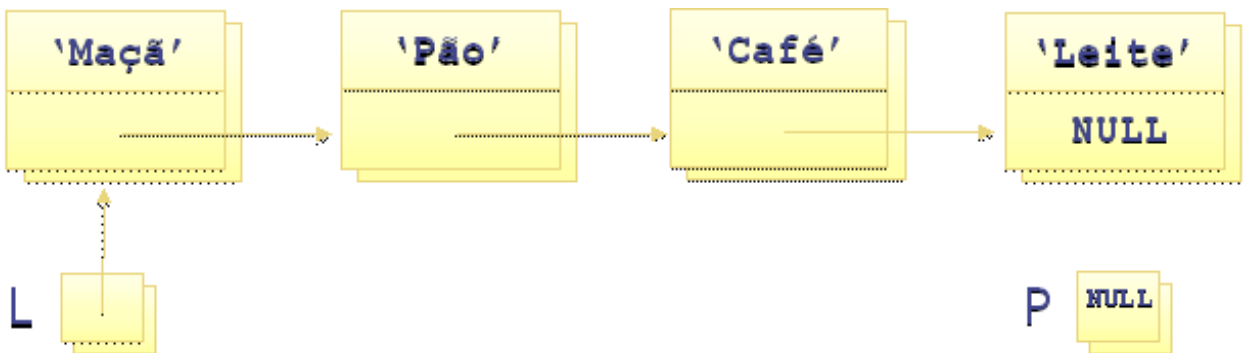
Recuperar item (dado uma chave)

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

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 (dado uma chave)

- **Pré-condição**: a lista não está vazia
- **Pós-condição**: remove um determinado item da lista dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

Remover item (dado uma chave)

```
1  int remover_item(LISTA_LIGADA *lista, int chave) {
2    NO *prem = lista->inicio;
3    NO *pant = NULL;
4
5    while(prem != NULL && prem->item->chave != chave) {
6      pant = prem;
7      prem = prem->proximo;
8    }
9
10   if(prem != NULL) {
11     if(prem == lista->inicio) {
12       lista->inicio = prem->proximo;
13     } else {
14       pant->proximo = prem->proximo;
15     }
16
17     if(prem == lista->fim) {
18       lista->fim = pant;
19     }
20
21     lista->tamanho--;
22     apagar_no(prem);
23     return 1;
24   }
25   return 0;
26 }
```

Exercício

Implementar as demais operações do TAD Listas

- Apagar 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

Introdução

- Das operações anteriores, a mais **complexa** é a **remoção** de um elemento dado uma chave

Introdução

- Das operações anteriores, a mais **complexa** é a **remoção** de um elemento dado uma chave
- 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

Introdução

- Das operações anteriores, a mais **complexa** é a **remoção** de um elemento dado uma chave
- 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**

Introdução

- Das operações anteriores, a mais **complexa** é a **remoção** de um elemento dado uma chave
- 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 lista_ligada LISTA_LIGADA;
2
3  struct lista_ligada {
4    NO *cabeca;
5    NO *fim;
6    int tamanho;
7  };
8
9  int vazia(LISTA_LIGADA *lista) {
10   return (lista->cabeca->proximo == NULL);
11 }
```

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

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 dado uma chave, já que não é necessário tratar separadamente quando o item a se remover é o primeiro

Remover item (dado uma chave)

```
1  int remover_item(LISTA_LIGADA *lista, int chave) {
2    if (!vazia(lista)) {
3      NO *paux = lista->cabeca;
4
5      while (paux->proximo != NULL &&
6             paux->proximo->item->chave != chave) {
7        paux = paux->proximo;
8      }
9
10     if (paux->proximo != NULL) {
11       NO *prem = paux->proximo;
12       paux->proximo = prem->proximo;
13
14       if (prem == lista->fim) {
15         lista->fim = paux;
16       }
17
18       apagar_no(prem);
19       lista->tamanho--;
20       return 1;
21     }
22   }
23   return 0;
24 }
```

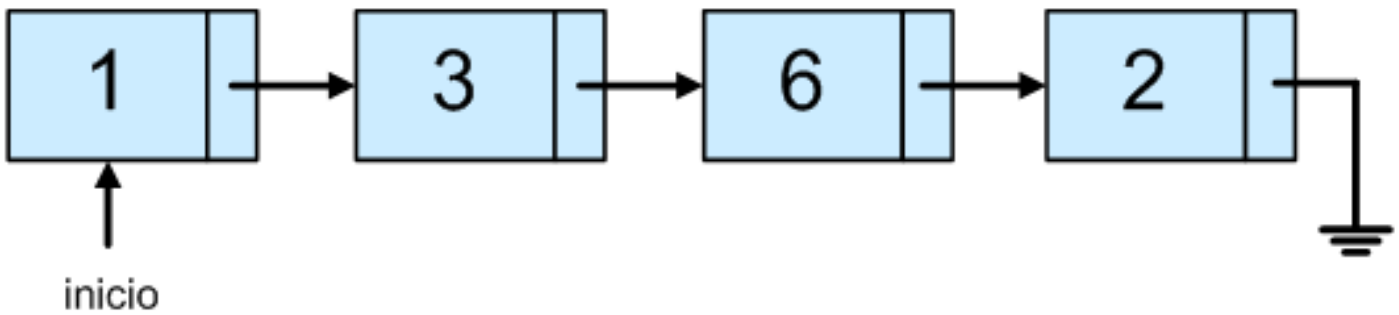
Exercício

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

Sumário

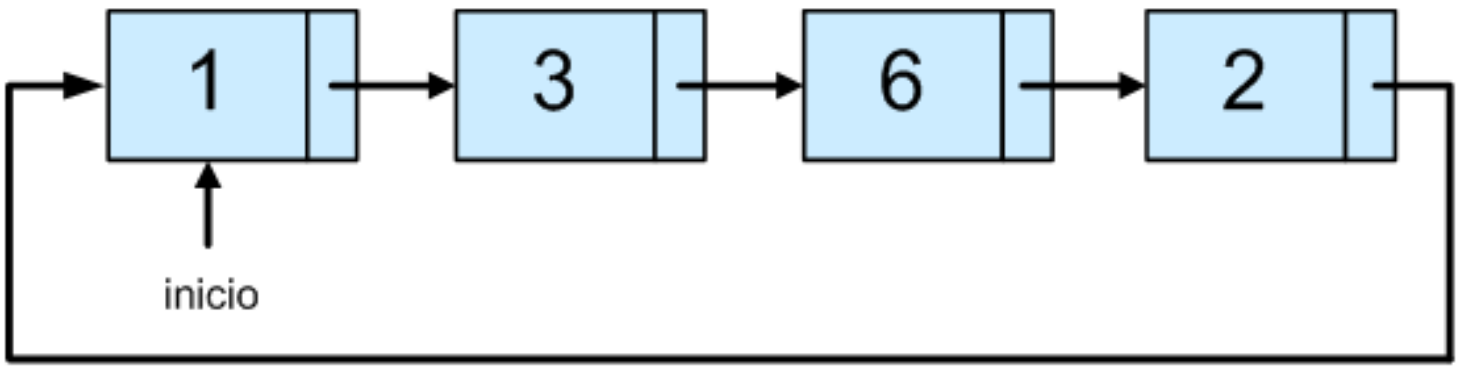
Listas Ligadas Circulares

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



Listas Ligadas Circulares

- A partir de um nó da lista pode-se chegar a qualquer outro nó

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

Exercício I

O problema de Josefo

Um pequeno exército se viu rodeado certa vez por um exército mais forte que ele. A única chance para não serem esmagados seria que alguém fosse buscar reforço montado no único cavalo da tropa. Para decidir quem seria o sortudo a ir buscar ajuda, decidiu-se colocar todos os soldados em um círculo, sorteando-se então um nome de um soldado e um número M. A partir do soldado sorteado, o M-ésimo soldado no sentido horário seria retirado da roda tendo que ficar no campo de batalha. Procedendo-se desta forma, o último soldado que restasse no círculo seria aquele que iria buscar ajuda.

Exercício II

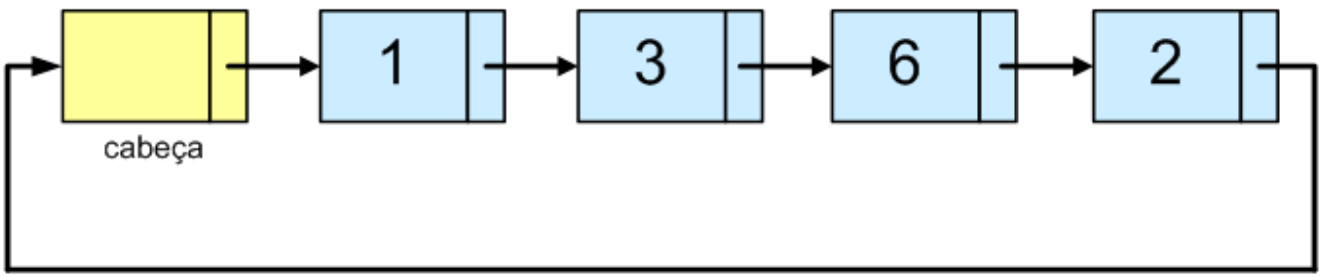
Exercício

- Implemente um procedimento que resolva o problema de Josefo. Esse procedimento deve receber como entrada uma lista e o número M e retornar o item encontrado.

```
1 ITEM *josefo(LISTA_LIGADA_CIRCULAR *lista, int M) {
2 ...
3 }
```

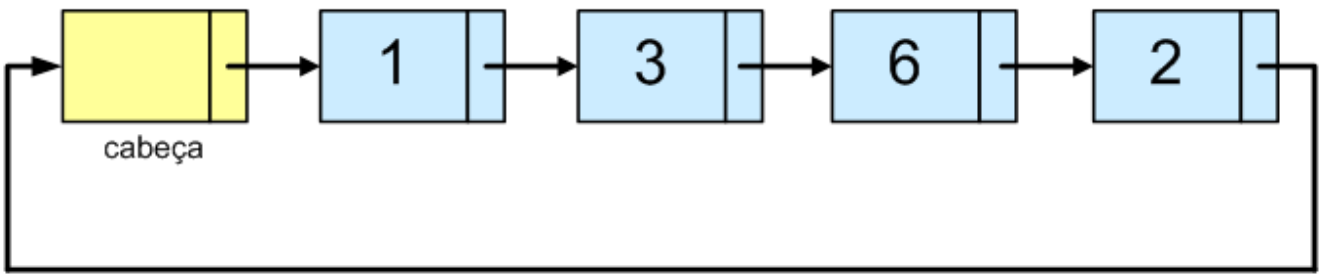
Listas Ligadas Circulares (Sentinela)

- No caso especial da busca em listas circulares, o emprego de um nó cabeça pode reduzir a quantidade de testes necessários



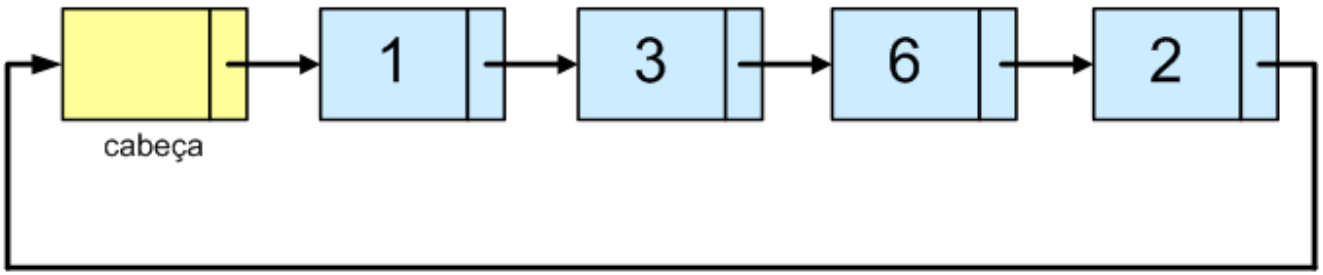
Listas Ligadas Circulares (Sentinela)

- 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



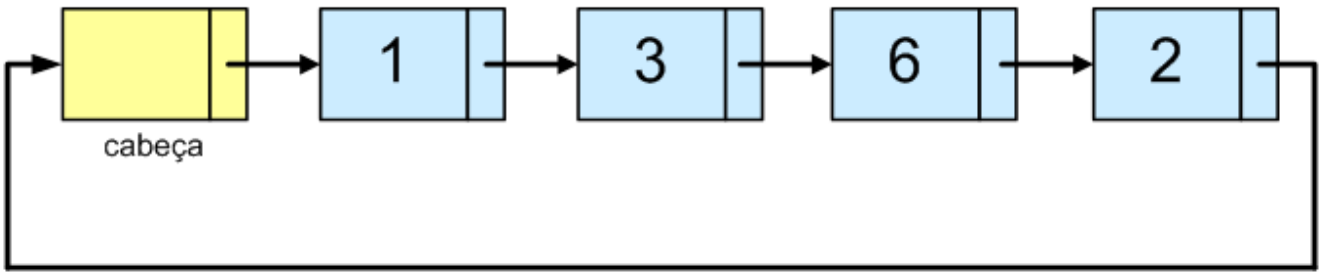
Listas Ligadas Circulares (Sentinela)

- 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



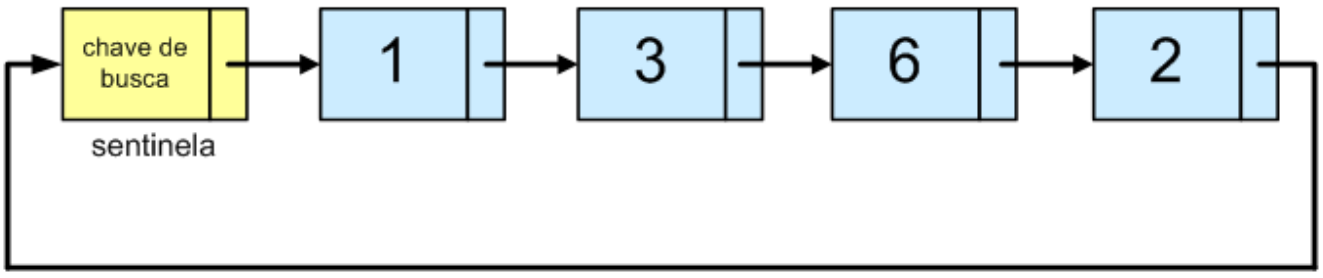
Listas Ligadas Circulares (Sentinela)

- 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**



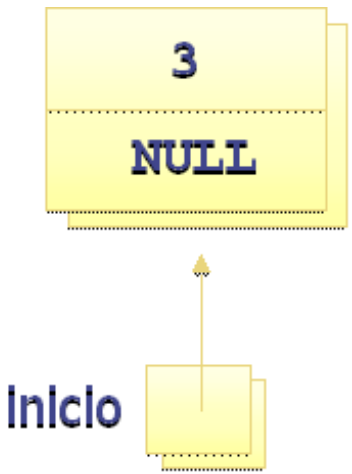
Listas Ligadas Circulares (Sentinela)

```
1  typedef struct lista_ligada LISTA_LIGADA;
2
3  struct lista_ligada {
4    NO *sentinela;
5    NO *fim;
6    int tamanho;
7  };
8
9
10 ITEM *recuperar_item(LISTA_LIGADA *lista, int chave) {
11   lista->sentinela->item->chave = chave;
12   NO *paux = lista->sentinela;
13
14   do {
15     paux = paux->proximo;
16   } while (paux->item->chave != chave);
17
18   return (paux != lista->sentinela) ? paux->item : NULL;
19 }
```

Sumário

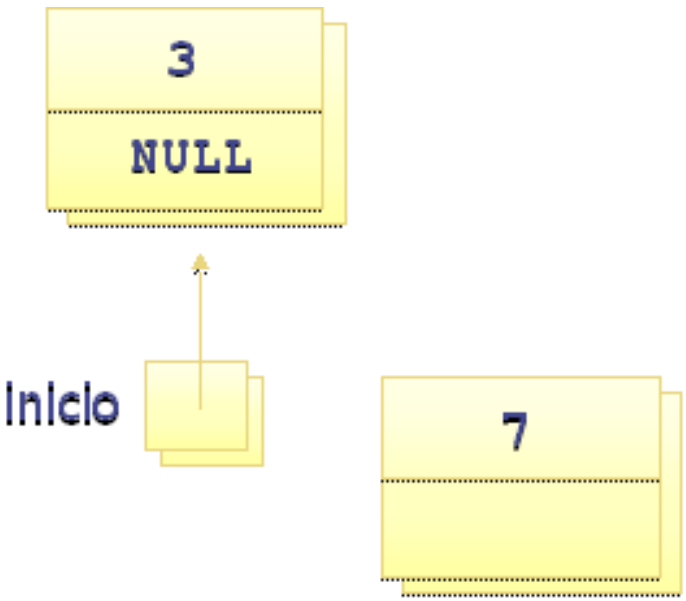
Inserção Ordenada - Lista Ligada

- Inserindo valor 3



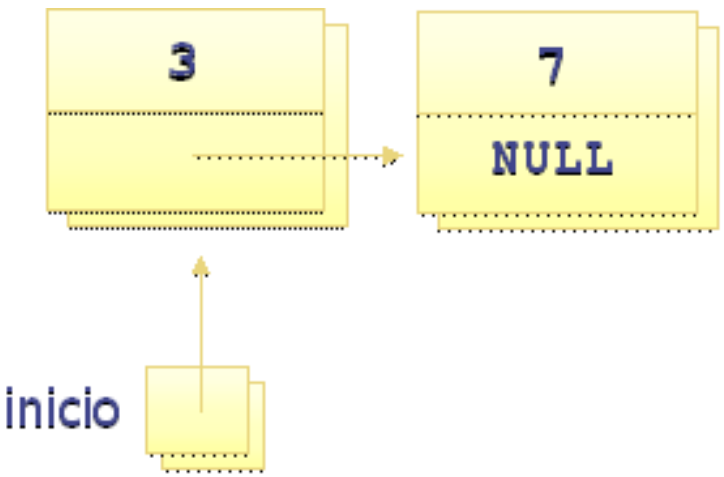
Inserção Ordenada - Lista Ligada

- Inserindo valor 7



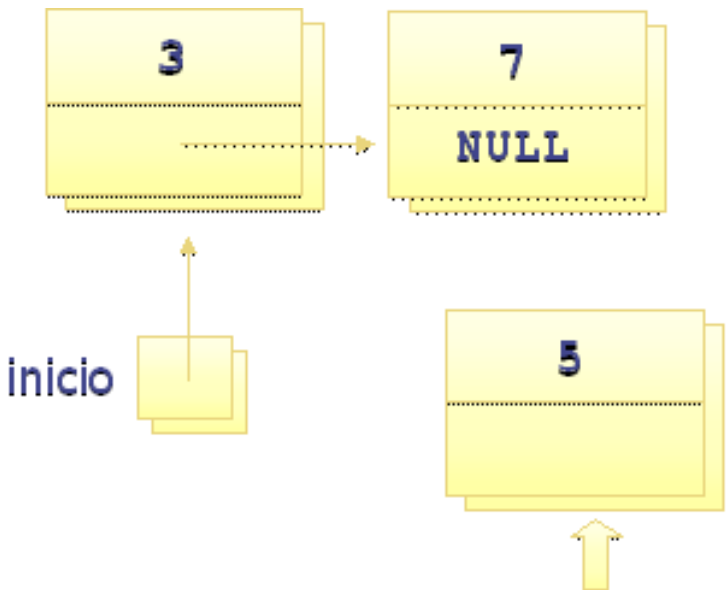
Inserção Ordenada - Lista Ligada

- Inserindo valor 7



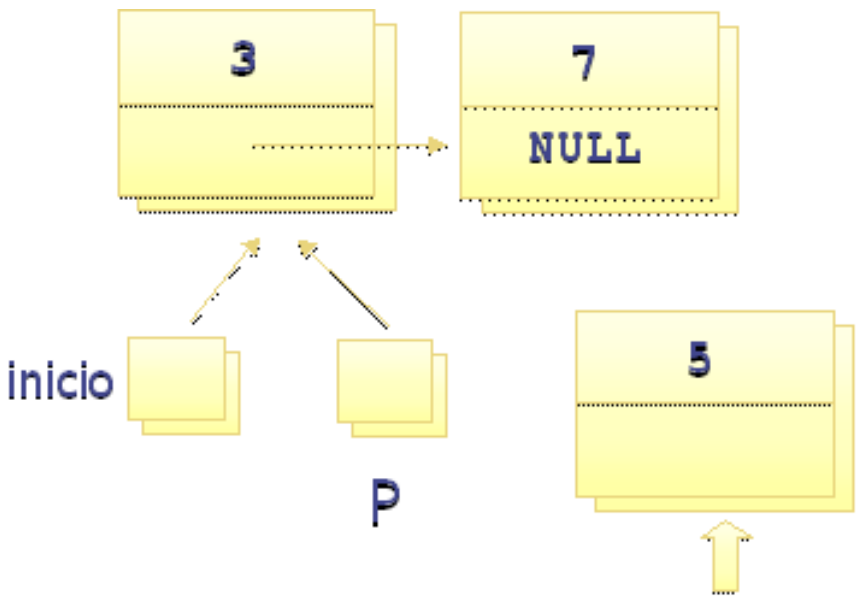
Inserção Ordenada - Lista Ligada

- Inserindo valor 5



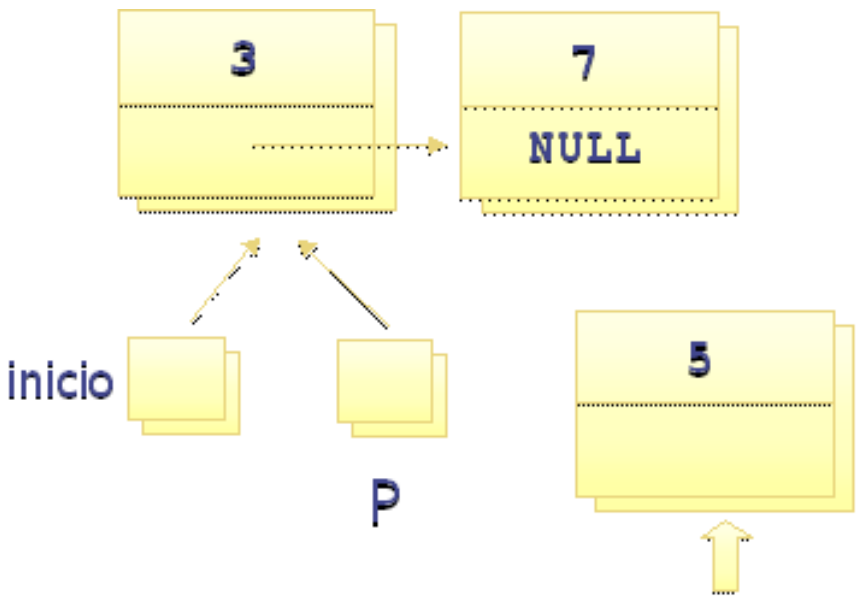
Inserção Ordenada - Lista Ligada

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



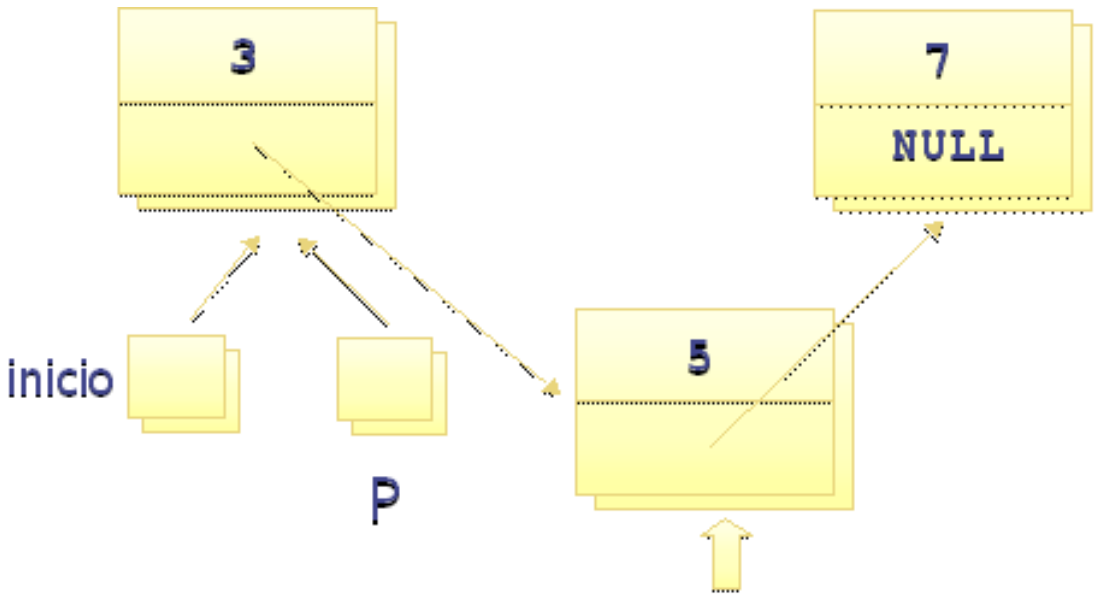
Inserção Ordenada - Lista Ligada

- Inserindo valor 5
- p->proximo.chave **maior** que novo.chave



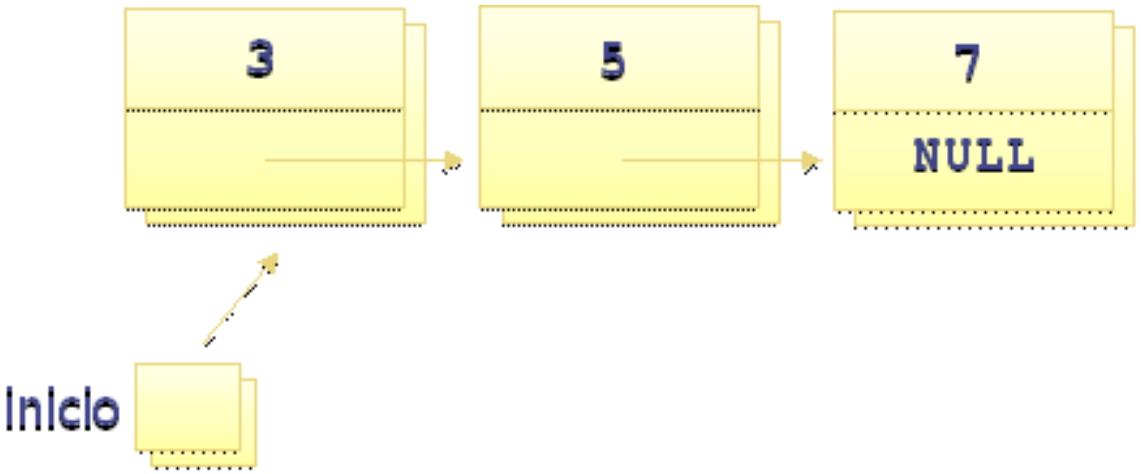
Inserção Ordenada - Lista Ligada

- Inserindo valor 5



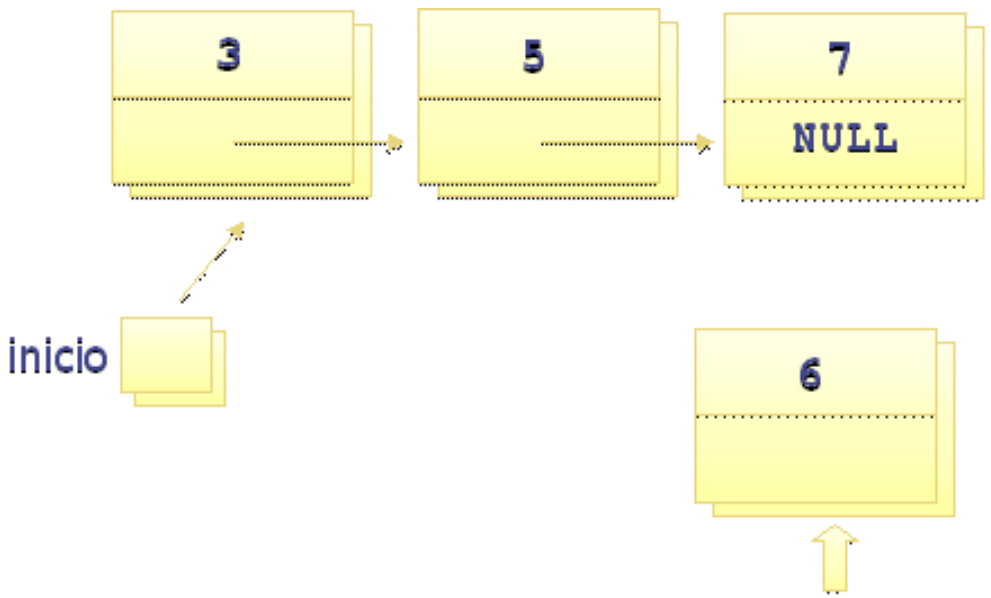
Inserção Ordenada - Lista Ligada

- Inserindo valor 5



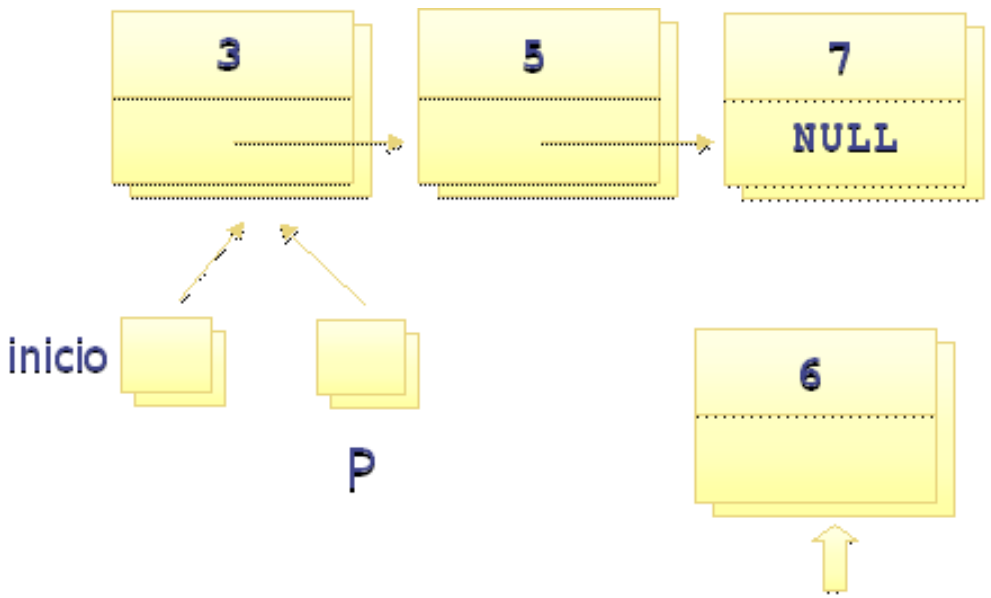
Inserção Ordenada - Lista Ligada

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



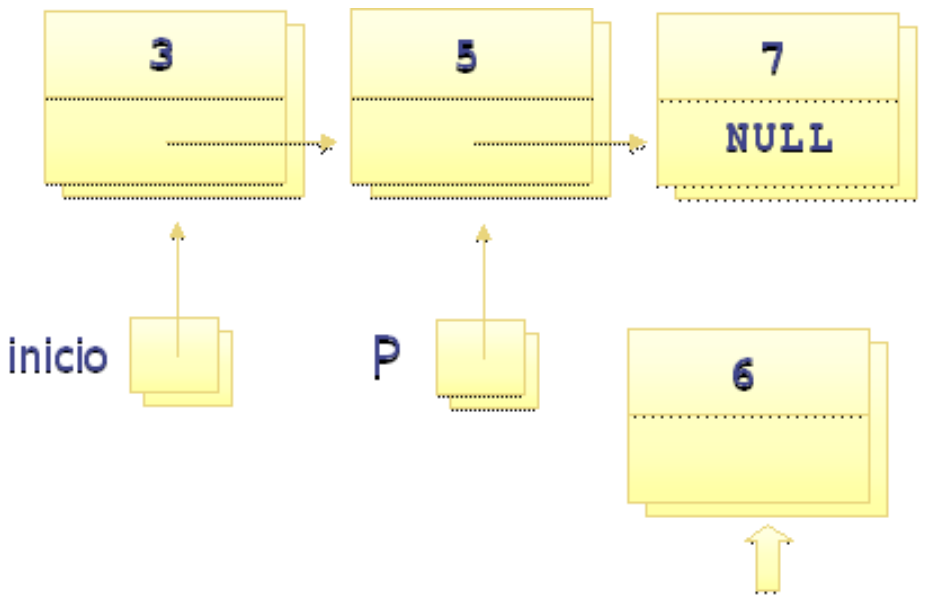
Inserção Ordenada - Lista Ligada

- Inserindo valor 6
- p->proximo.chave **menor** que novo.chave



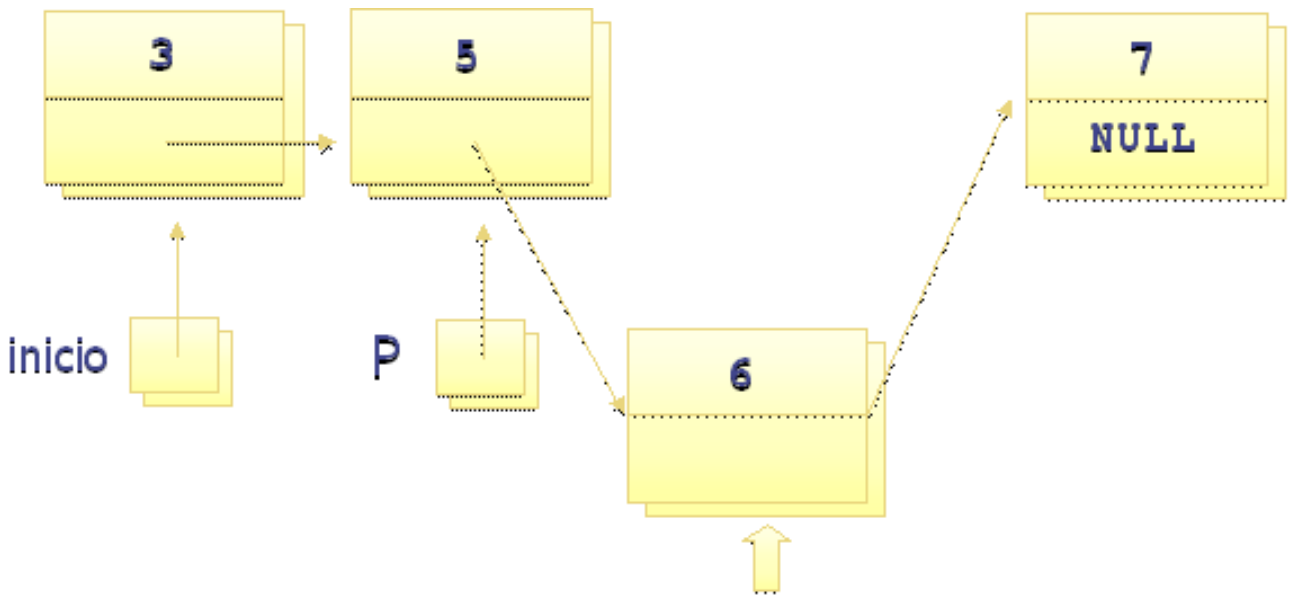
Inserção Ordenada - Lista Ligada

- Inserindo valor 6
- p->proximo.chave **maior** que novo.chave



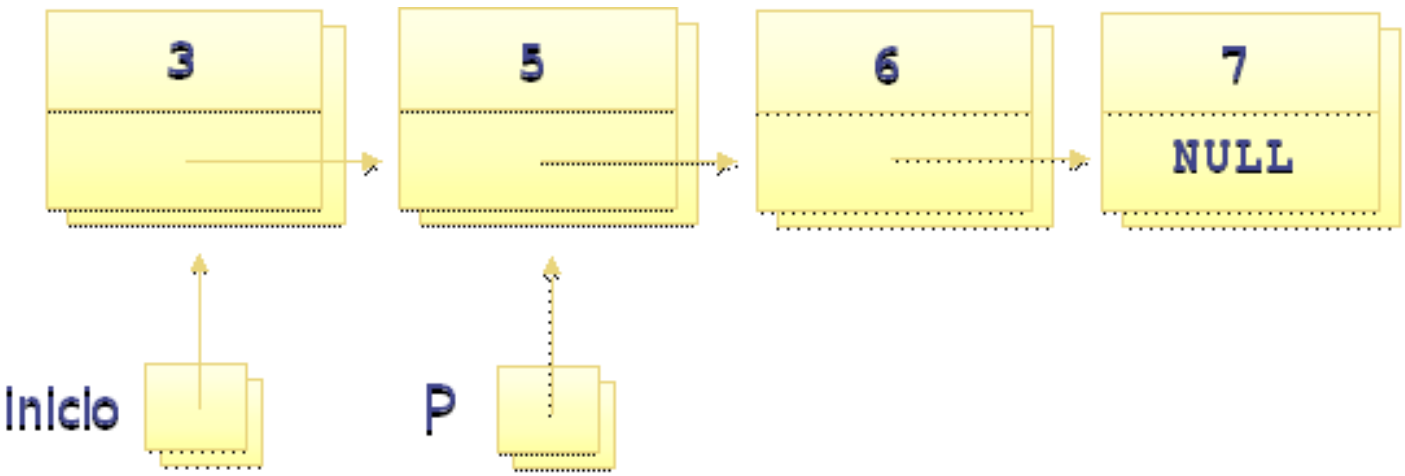
Inserção Ordenada - Lista Ligada

- Inserindo valor 6
- p->proximo.chave **maior** que novo.chave



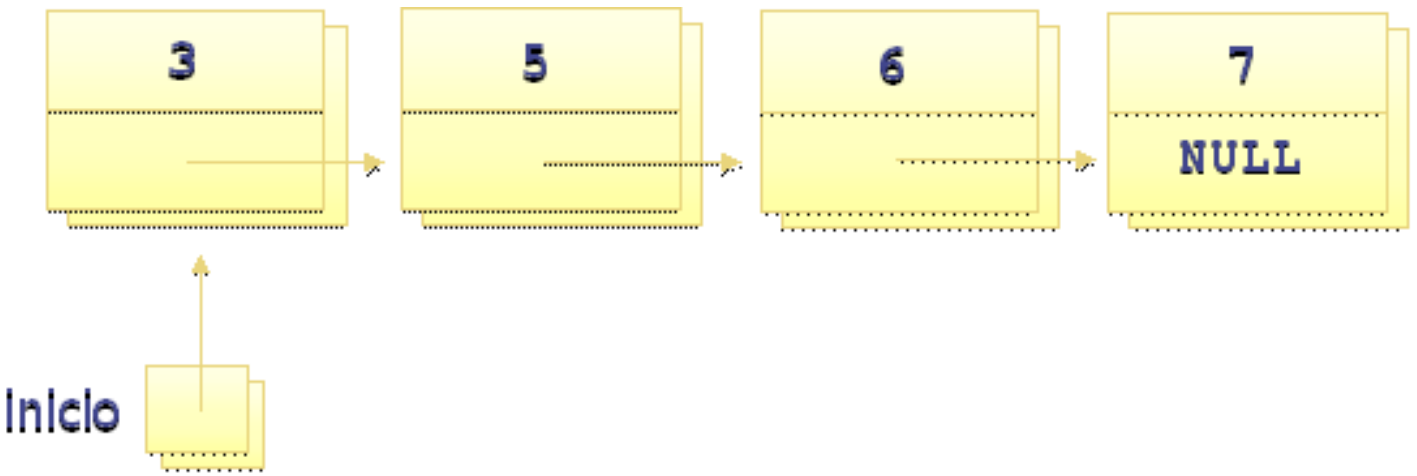
Inserção Ordenada - Lista Ligada

- Inserindo valor 6



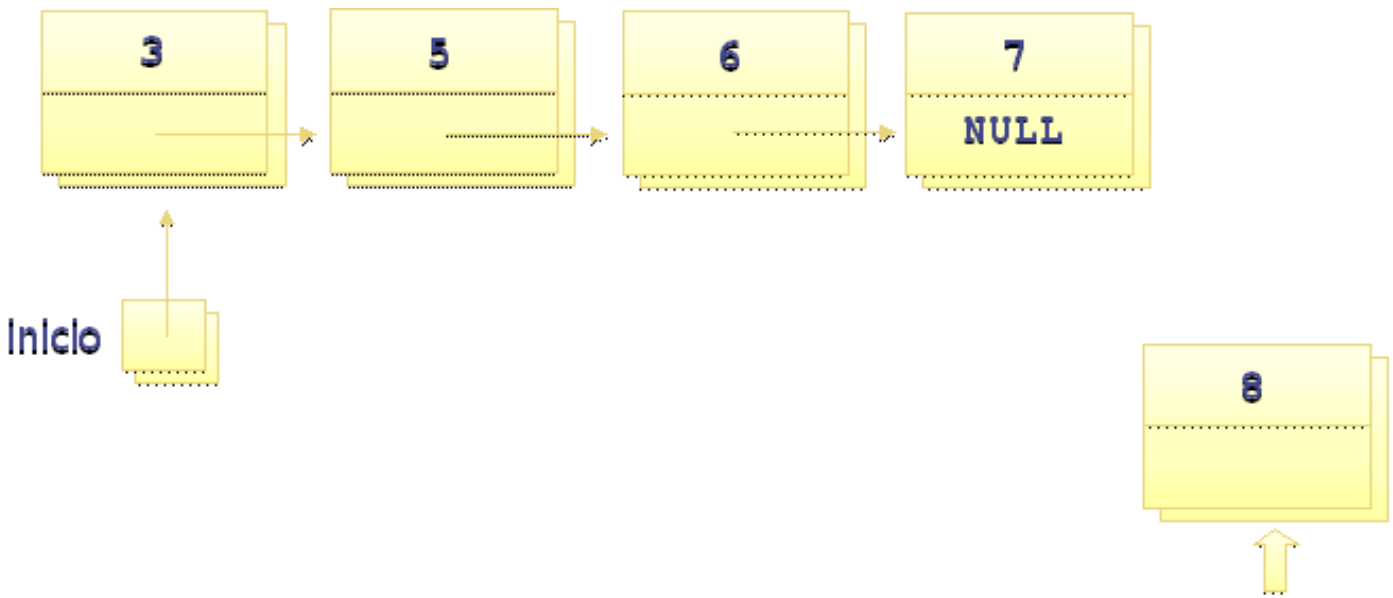
Inserção Ordenada - Lista Ligada

- Inserindo valor 6



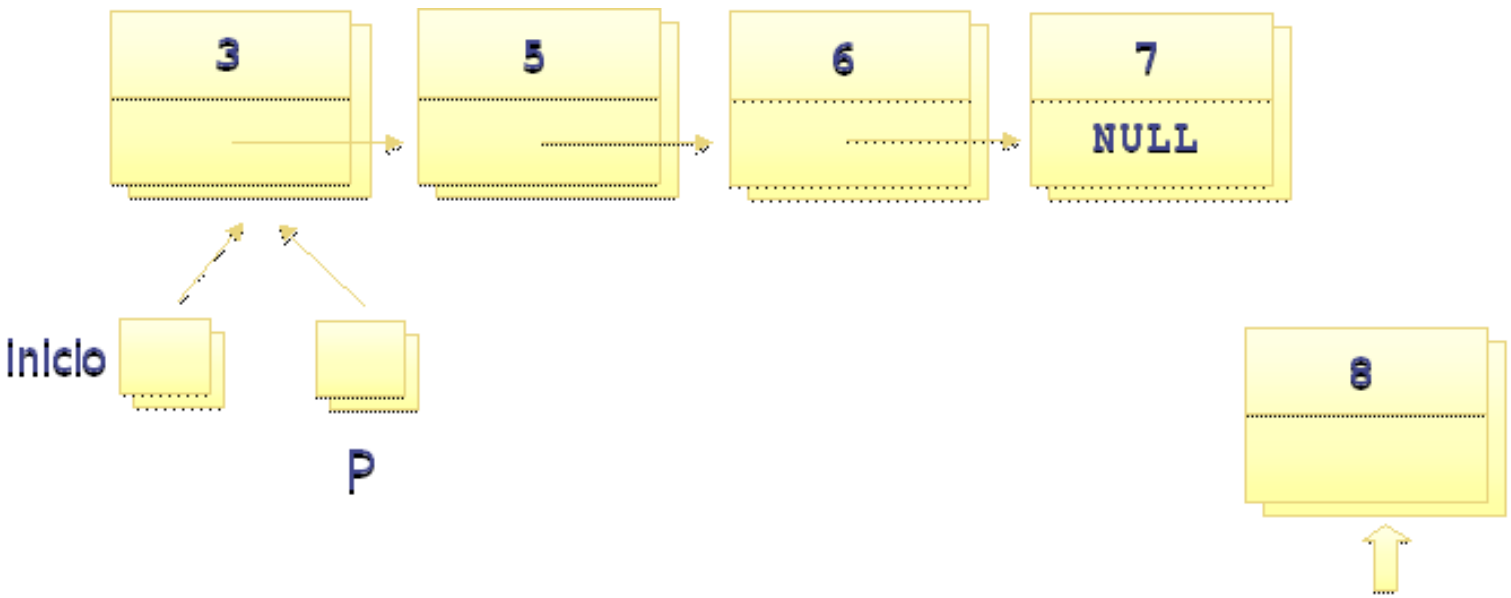
Inserção Ordenada - Lista Ligada

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



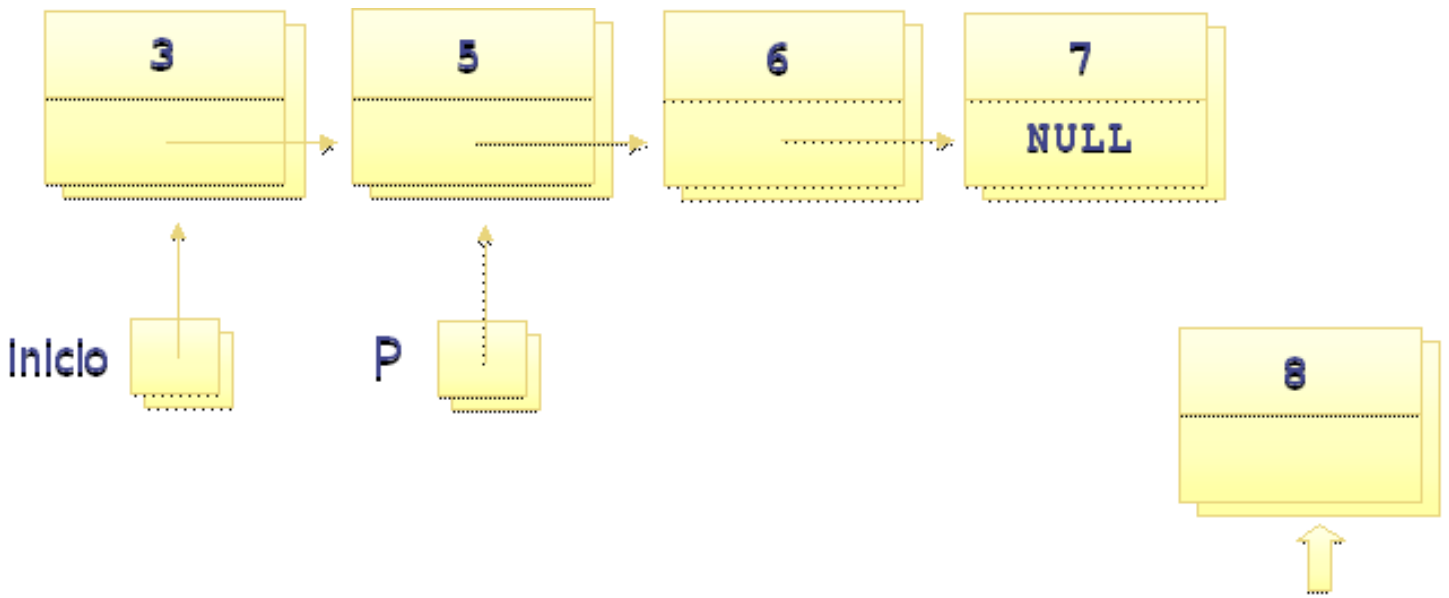
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- p->proximo.chave **menor** que novo.chave



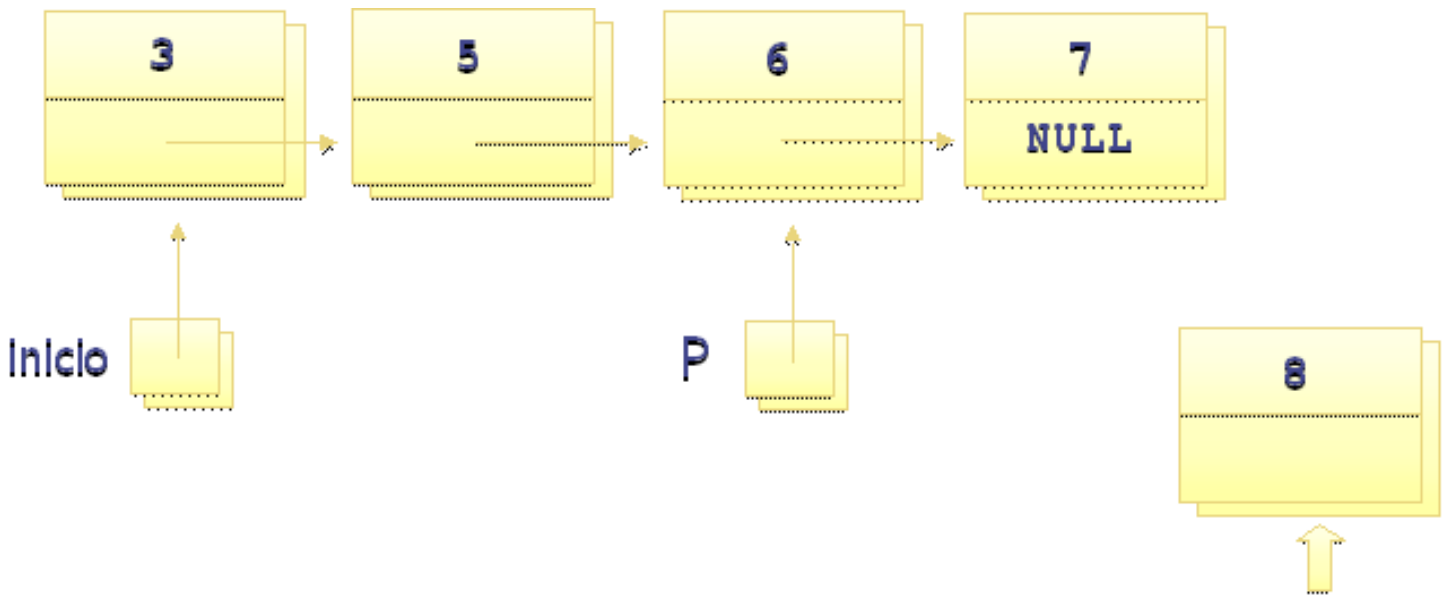
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- p->proximo.chave **menor** que novo.chave



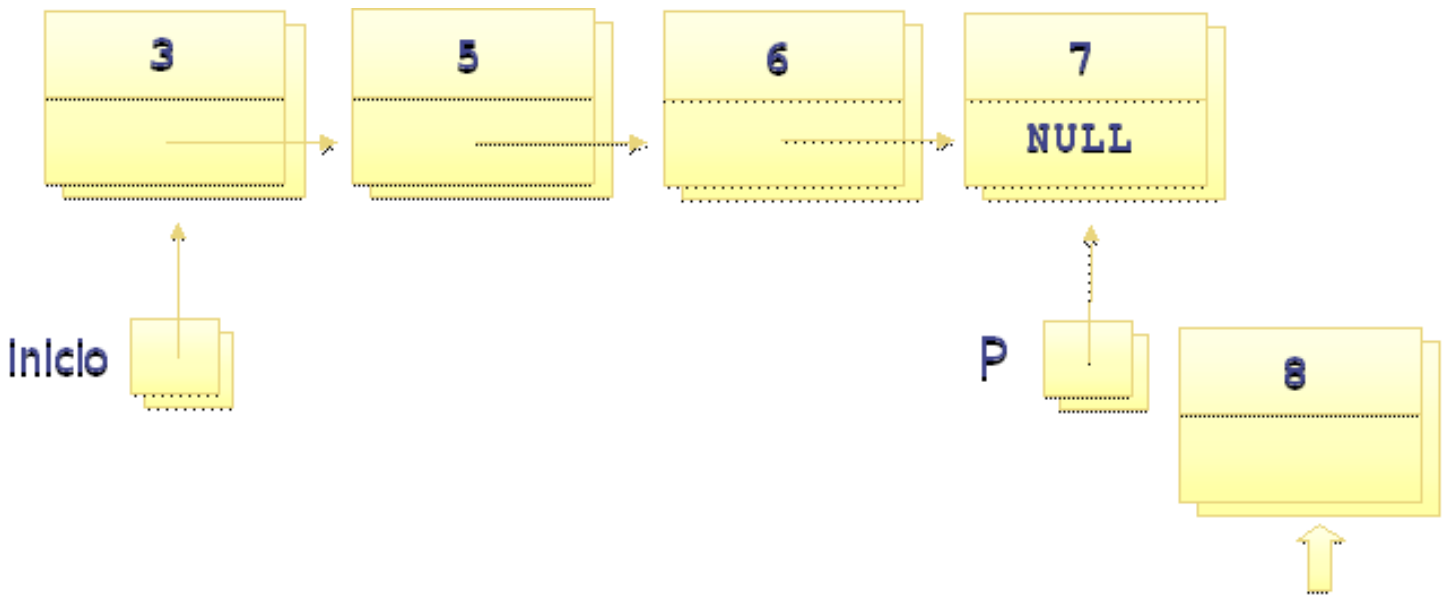
Inserção Ordenada - Lista Ligada

- Inserindo valor 8
- p->proximo.chave **menor** que novo.chave



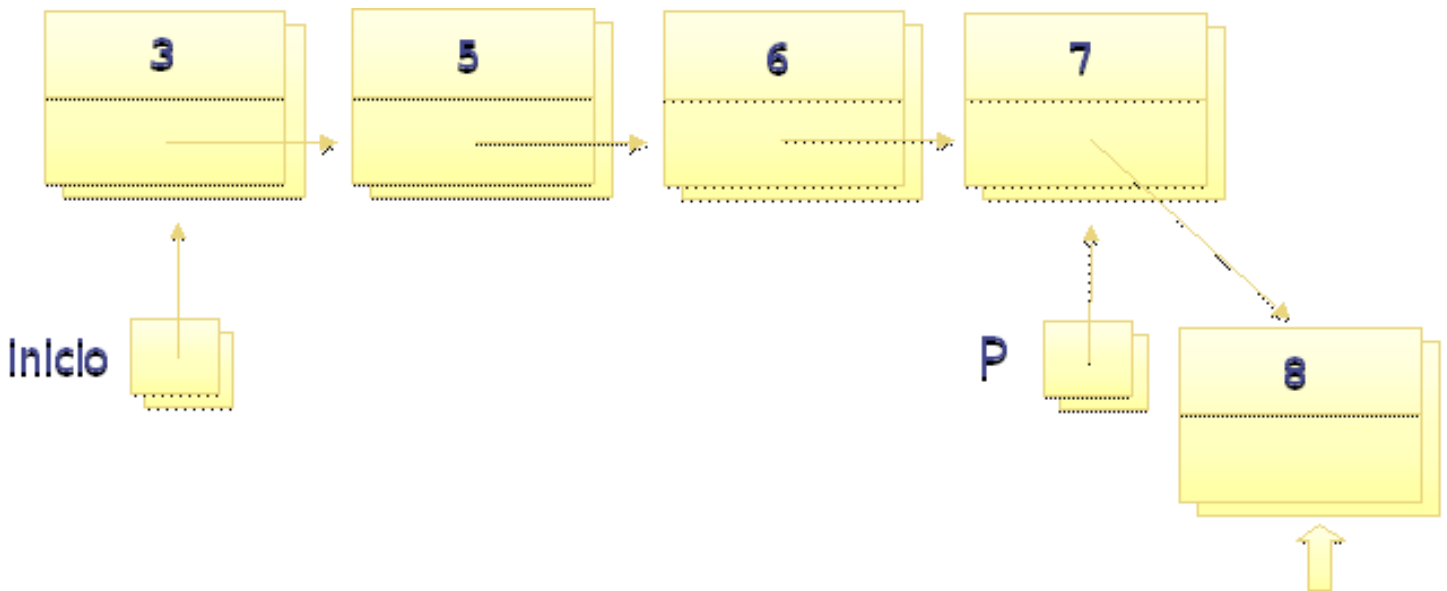
Inserção Ordenada - Lista Ligada

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



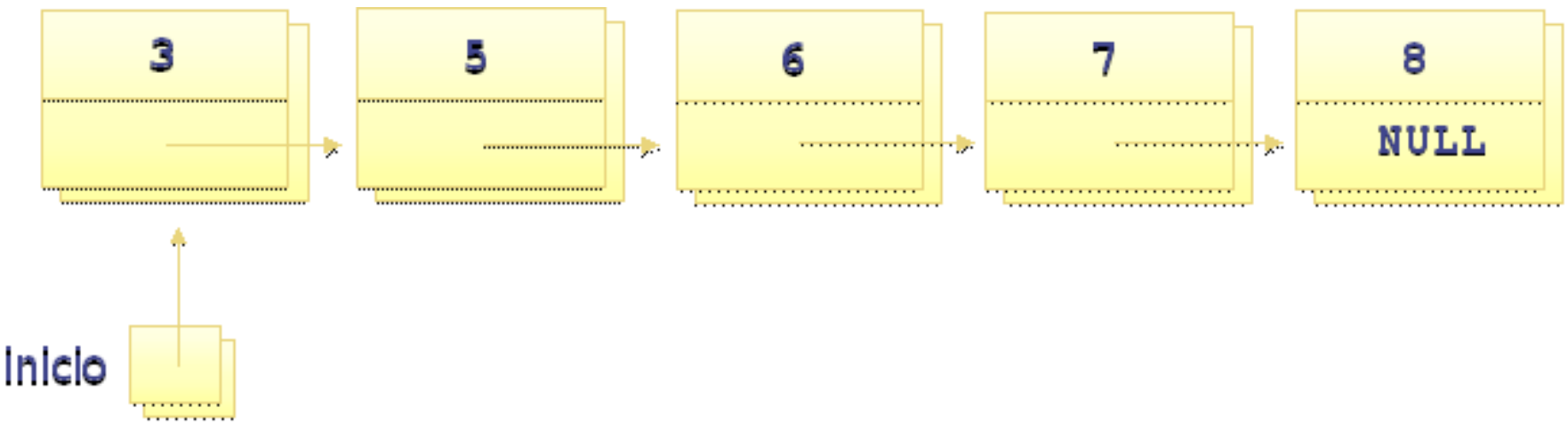
Inserção Ordenada - Lista Ligada

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



Inserção Ordenada - Lista Ligada

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



Comentários sobre a Implementação

- Não precisa do ponteiro fim porque a inserção será em qualquer posição de lista

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_item(LISTA_LIGADA *lista, ITEM *item) {
2    NO *pnovo = (NO *) malloc(sizeof (NO));
3
4    if (pnovo != NULL) {
5      pnovo->item = item;
6      pnovo->proximo = NULL;
7
8      NO *paux = lista->cabeca;
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     lista->tamanho++;
18
19     return 1;
20   } else {
21     return 0;
22   }
23 }
```

Busca em Lista Ordenada

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

```
1  ITEM *recuperar_item(LISTA_LIGADA *lista, int chave) {
2    if (!vazia(lista)) {
3      NO *paux = lista->cabeca->proximo;
4
5      while (paux != NULL) {
6        if (paux->item->chave == chave) {
7          return paux->item;
8        } else if (paux->item->chave > chave) {
9          return 0;
10       }
11       paux = paux->proximo;
12     }
13   }
14   return NULL;
15 }
```

Lista Ordenada - Outras Operações

- As demais operações implementadas podem deixar a lista desordenada?

Lista Ordenada - Outras Operações

- As demais operações implementadas podem deixar a lista desordenada?
- Poderia ocorrer com a remoção de elementos, entretanto

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

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