

# LISTAS LIGADAS

**Prof.: Julio Cesar dos Reis**

# Roteiro

2

- ❑ Deficiências das listas com arranjos
- ❑ Definição de lista ligada
- ❑ Modelagem da estrutura ligada
- ❑ Operações sobre lista ligada

# Deficiências das listas com arranjos

3

- Gerenciamento de memória
  - ▣ Arranjos de tamanho fixo alocados contiguamente
    - Tamanho pequeno pode faltar espaço
    - Tamanho grande pode desperdiçar memória
  - ▣ Arranjos dinâmicos não resolvem por completo o problema

# Deficiências das listas com arranjos

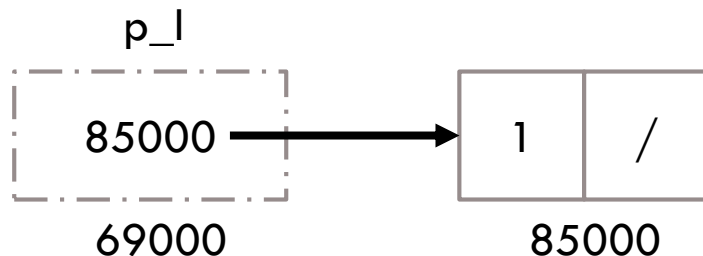
4

- Gerenciamento de memória
  - ▣ Arranjos de tamanho fixo alocados contiguamente
    - Tamanho pequeno pode faltar espaço
    - Tamanho grande pode desperdiçar memória
  - ▣ Arranjos dinâmicos não resolvem por completo o problema
- Problemas na inserção e remoção (ordenada)
  - ▣ Necessidade de deslocar elementos

# Estrutura ligada

5

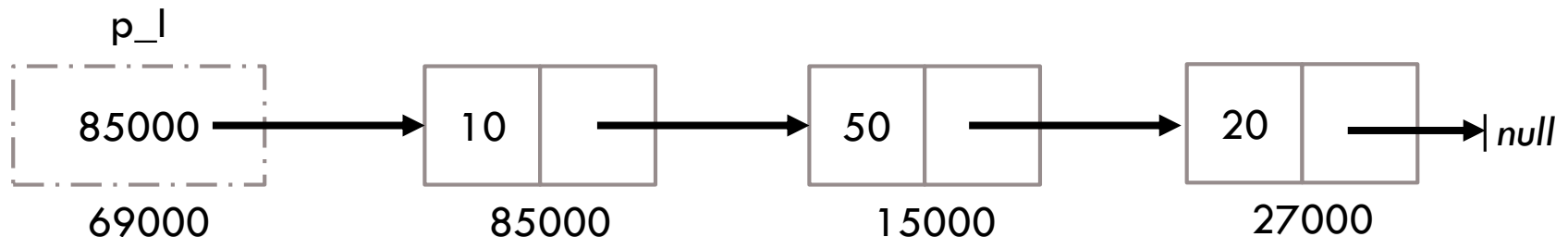
- Pode-se alocar e desalocar a memória para os elementos sob demanda
  - ▣ Uma variável ponteiro indica o primeiro elemento da estrutura



# Estrutura ligada

6

- Pode-se alocar e desalocar a memória para os elementos sob demanda
  - ▣ Uma variável ponteiro indica o primeiro elemento da estrutura
  - ▣ Cada elemento indica seu sucessor
    - Primeiro elemento aponta para o segundo, e assim sucessivamente



# Definição de Lista Ligada

7

- Conjunto em que cada elemento é representado por um nó contendo uma ligação para outro nó
  - ▣ Estrutura linear
    - Cada elemento possui no máximo um predecessor e um sucessor

# Definição de Lista Ligada

8

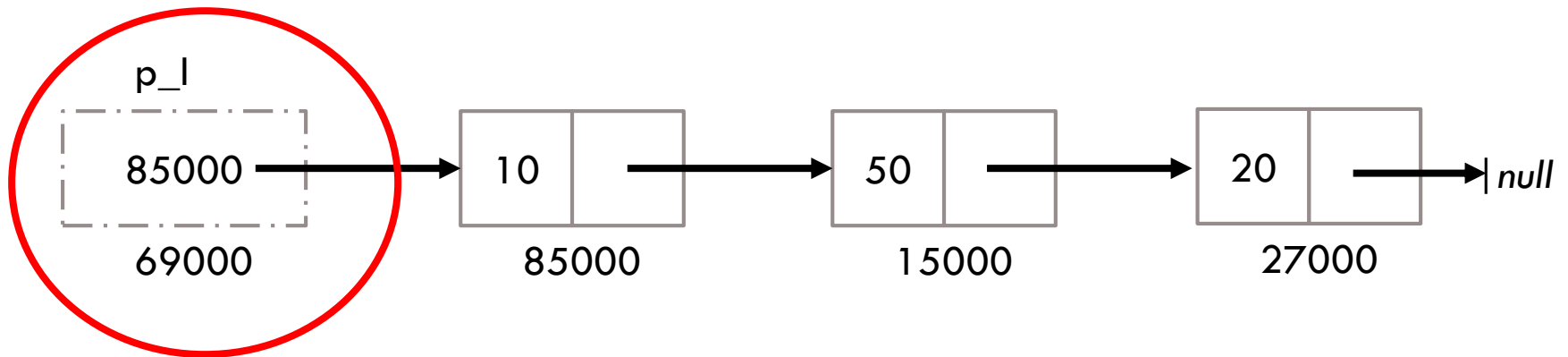
- Conjunto em que cada elemento é representado por um nó contendo uma ligação para outro nó
  - ▣ Estrutura linear
    - Cada elemento possui no máximo um predecessor e um sucessor
- Nó é o elemento alocado dinamicamente e contém
  - ▣ Armazenamento dos dados
  - ▣ Ponteiro para o nó subsequente



# Lista ligada

9

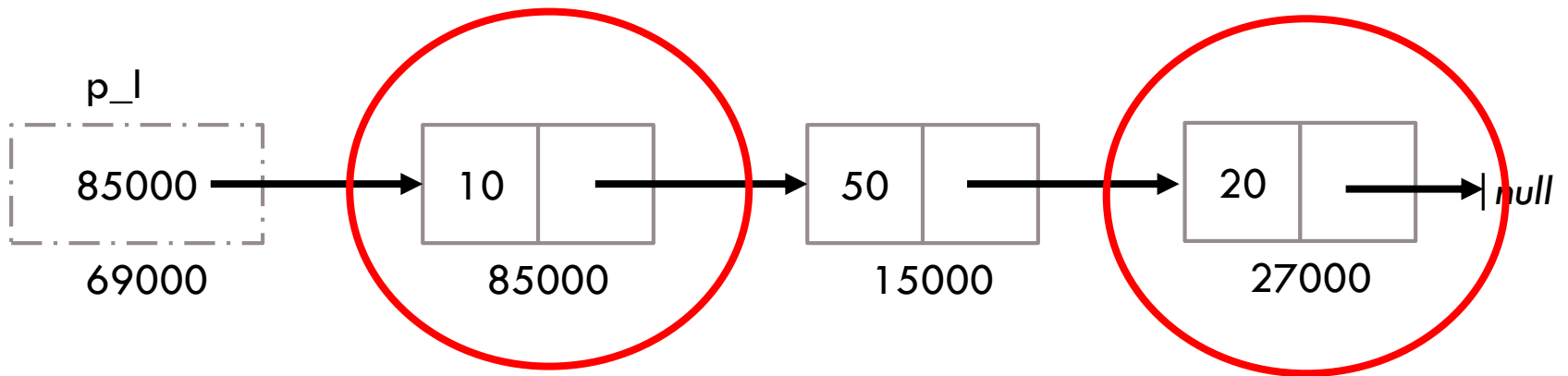
## □ Ponteiro inicial



# Lista ligada

10

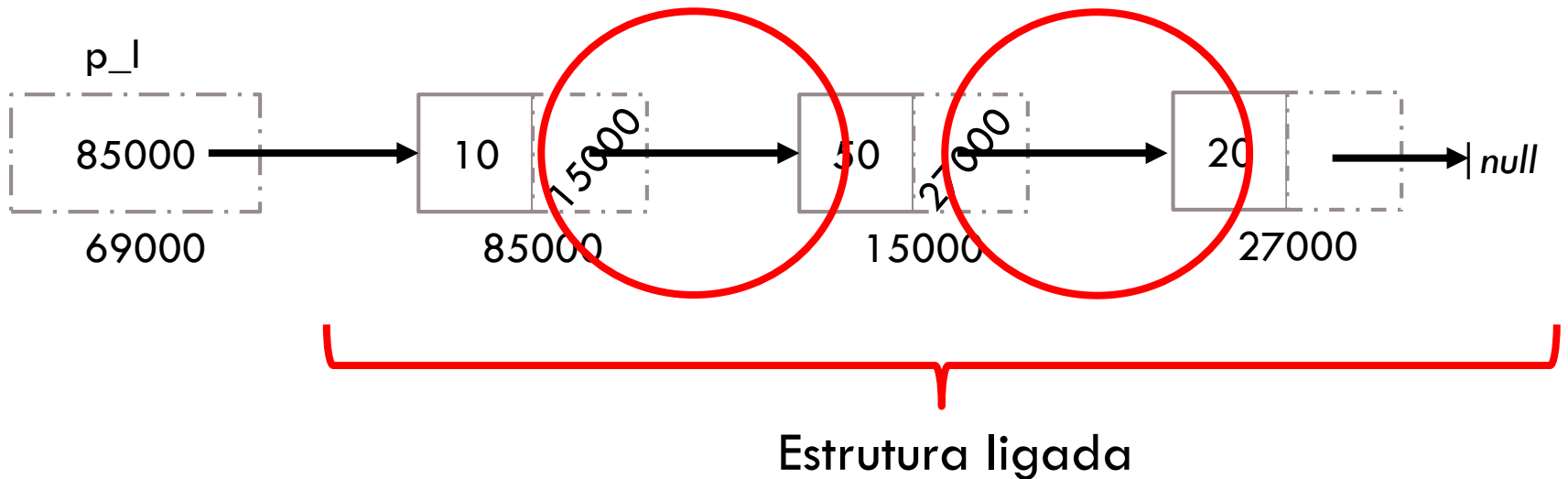
## □ Nós



# Lista ligada

11

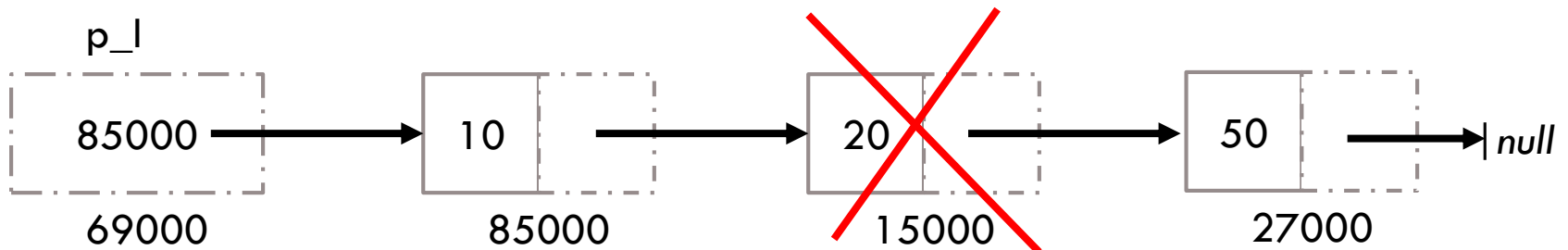
## □ Ligação entre os nós (ponteiros)



# Exemplo de manipulação - remoção

12

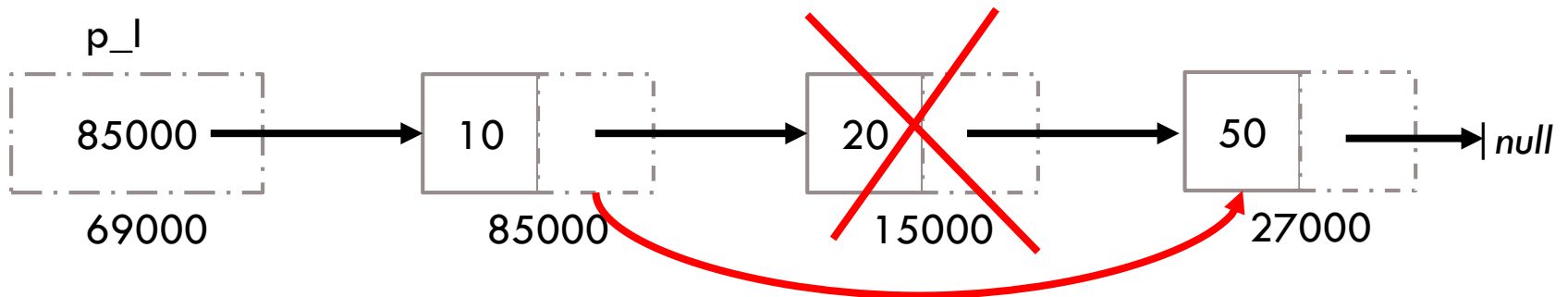
## □ Remover o elemento 20



# Exemplo de manipulação - remoção

13

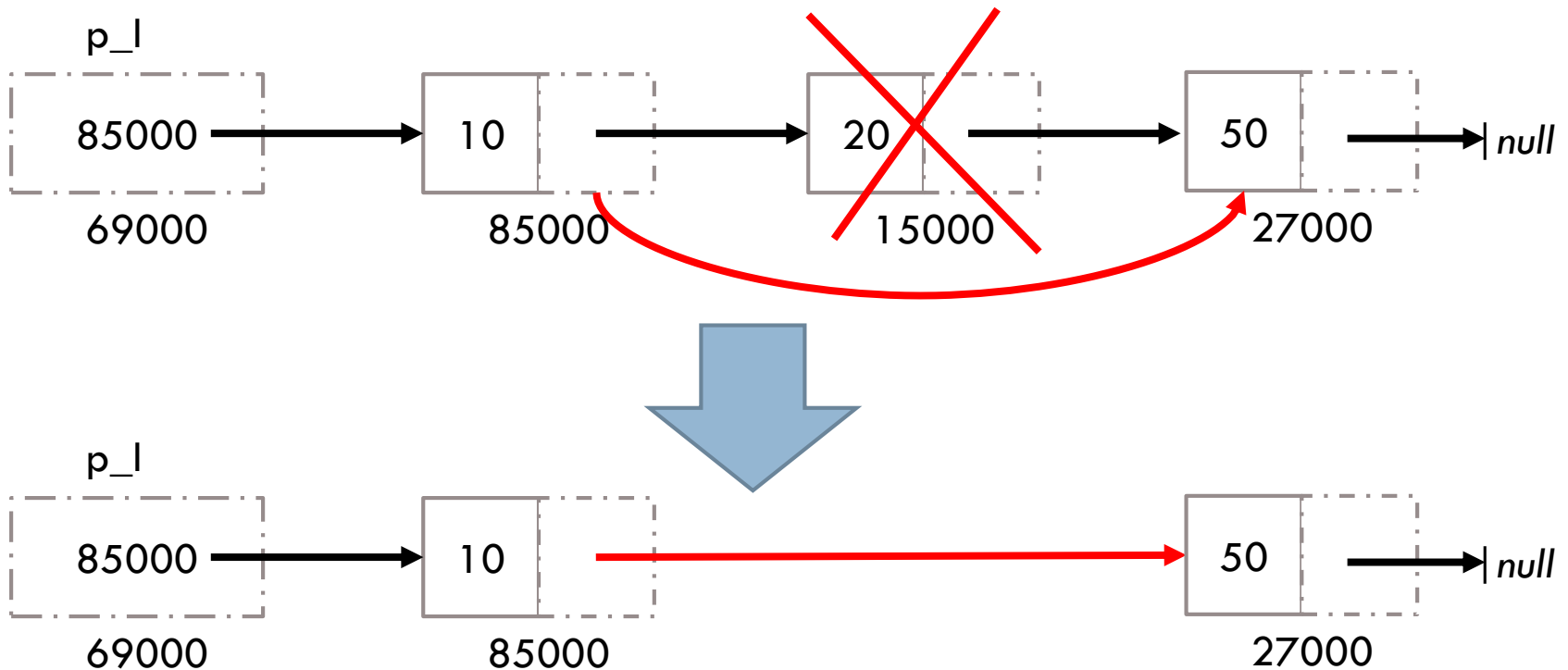
## □ Remover o elemento 20



# Exemplo de manipulação - remoção

14

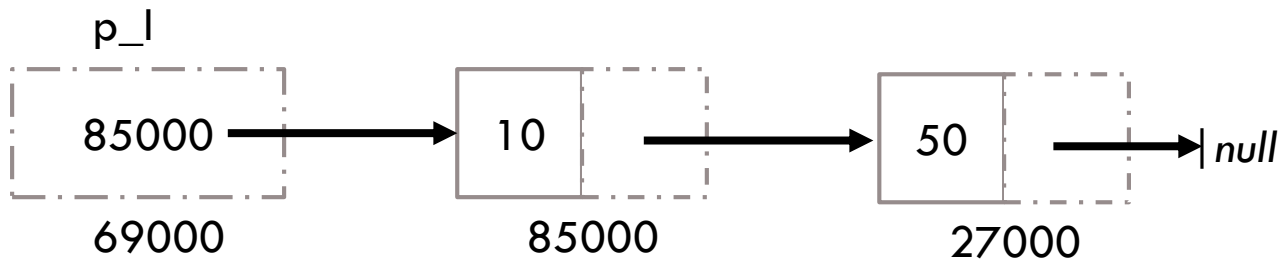
## □ Remover o elemento 20



# Exemplo de manipulação - inserção

15

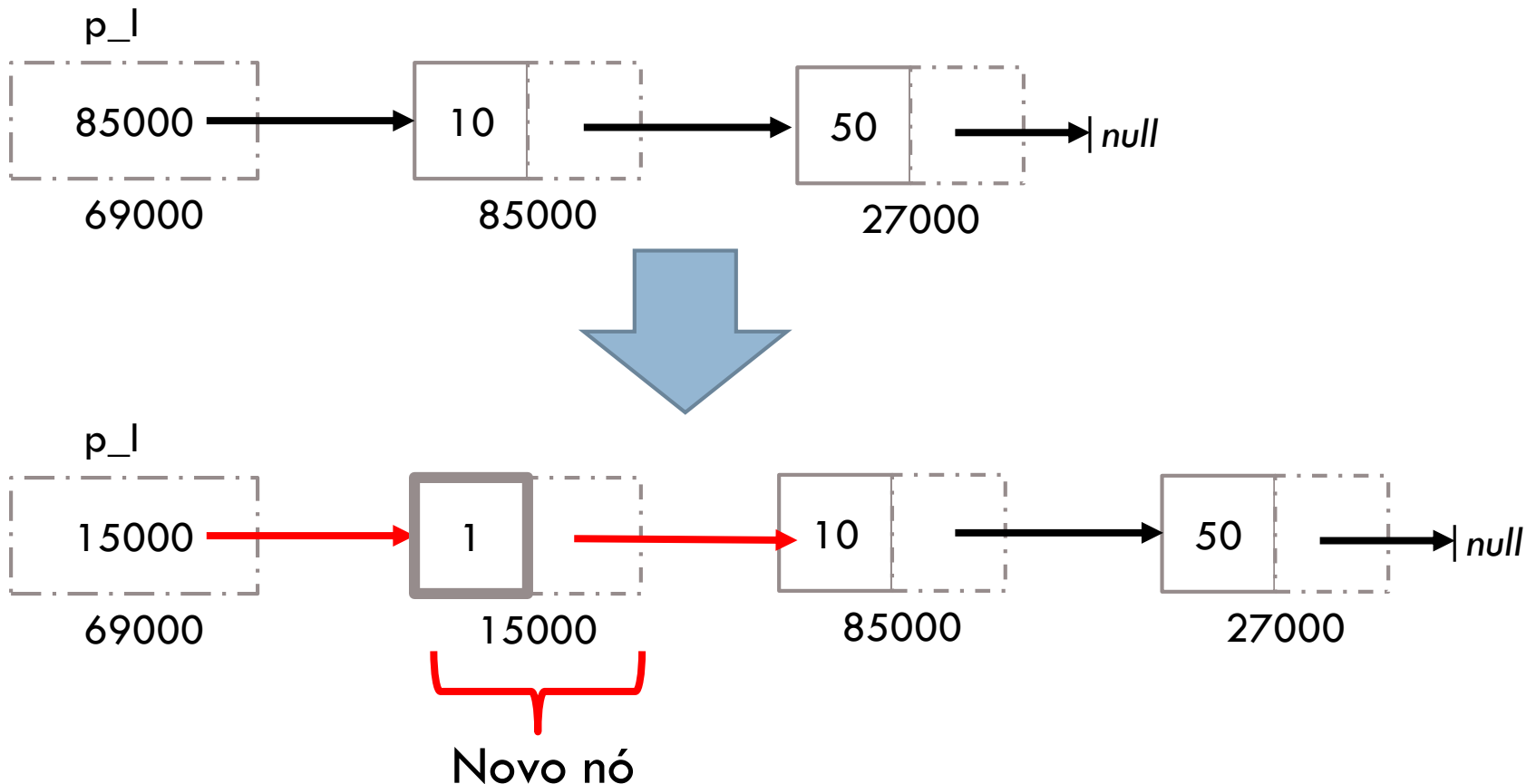
## □ Inserção do elemento 1 (ordenado)



# Exemplo de manipulação - inserção

16

## □ Inserção do elemento 1 (ordenado)





# Definição das estruturas

17

## Elemento nó

```
typedef struct No{  
    int chave;  
    // outros campos...  
    struct No *prox;  
} No;
```

Ponteiro para o próximo elemento Nó  
- Usa-se **struct No** dentro da estrutura pois o tipo **No** ainda não foi definido

# Definição das estruturas

18

## Elemento nó

```
typedef struct No{  
    int chave;  
    // outros campos...  
    struct No *prox;  
} No;
```

Ponteiro para o próximo elemento Nó  
- Usa-se **struct No** dentro da estrutura pois o tipo **No** ainda não foi definido

## Lista ligada


```
typedef struct {  
    No *inicio;  
    // outros campos...  
} Lista;
```

# Outras opções sobre as estruturas

19

## Elemento nó

```
typedef struct No{  
    RegLista dados;  
    struct No *prox;  
} No;
```



```
typedef struct {  
    int chave;  
    // outros campos...  
} RegLista;
```


# Outras opções sobre as estruturas

20

## Elemento nó

```
typedef struct No{  
    RegLista dados;  
    struct No *prox;  
} No;
```

```
typedef struct {  
    int chave;  
    // outros campos...  
} RegLista;
```



## Lista ligada

```
typedef struct {  
    No *inicio;  
} Lista;
```

No \*p\_lista;



typedef struct No\* LISTA;  
E.g.: LISTA p; // é um ponteiro



# Operações sobre Lista Ligada

21

- Inicialização

- **Inserção**

- No início e no meio da lista desordenada
- Elemento com chave indicada pelo usuário (lista ordenado)

# Operações sobre Lista Ligada

22

- Inicialização
- **Inserção**
  - ▣ No início e no meio da lista desordenada
  - ▣ Elemento com chave indicada pelo usuário (lista ordenado)
- **Remoção**
  - ▣ No início e no meio da lista
  - ▣ Elemento com chave indicada pelo usuário
- Impressão e destruição

# Inicialização

23

- Fazer campo inicio apontar para *null*

```
void iniciar_lista(Lista *p_l){  
    p_l->inicio=null;  
}
```

## Código cliente

```
Lista l;  
iniciar_lista(&l);
```

# Inicialização

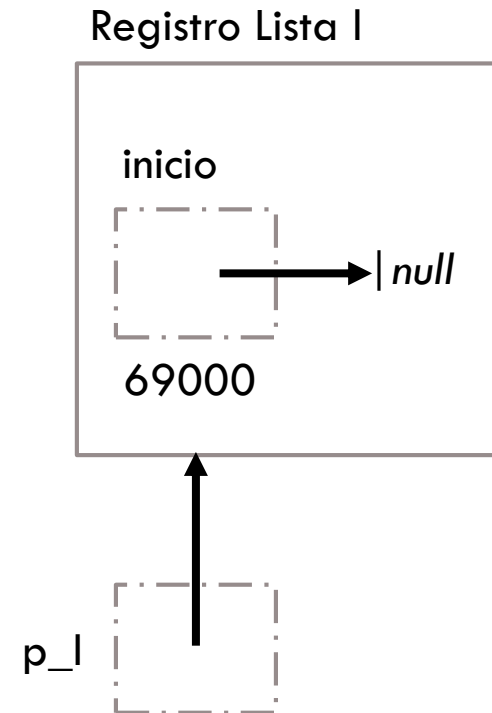
24

- Fazer campo inicio apontar para *null*

```
void iniciar_lista(Lista *p_l){  
    p_l->inicio=null;  
}
```

## Código cliente

```
Lista l;  
iniciar_lista(&l);
```





25

## Algoritmos de inserção em lista ligada

# Inserção (no início da lista)

26

## □ Aloca um novo nó

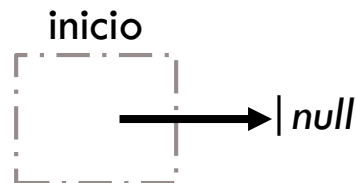
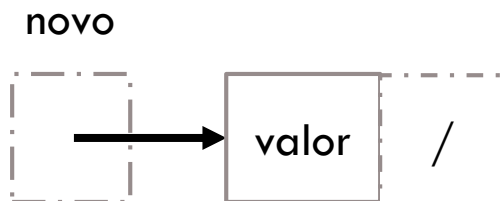
```
void insercao_inicio_lista(Lista *p_l, int valor){  
    No *novo = malloc(sizeof(No));  
    novo->chave = valor;  
    (...)  
}
```

# Inserção (no início da lista)

27

## □ Aloca um novo nó

```
void insercao_inicio_lista(Lista *p_l, int valor){  
    No *novo = malloc(sizeof(No));  
    novo->chave = valor;  
    (...)  
}
```



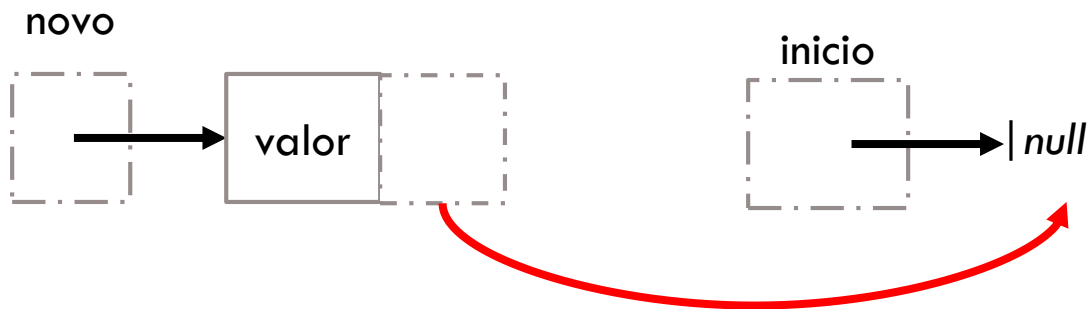
Lista existente (vazia)

# Inserção (no início da lista)

28

- Campo *prox* do novo aponta para o mesmo que o *inicio*

```
void insercao_inicio_lista(Lista *p_l, int valor){  
    No *novo = malloc(sizeof(No));  
    novo->chave = valor;  
    novo->prox = p_l->inicio;  
}
```

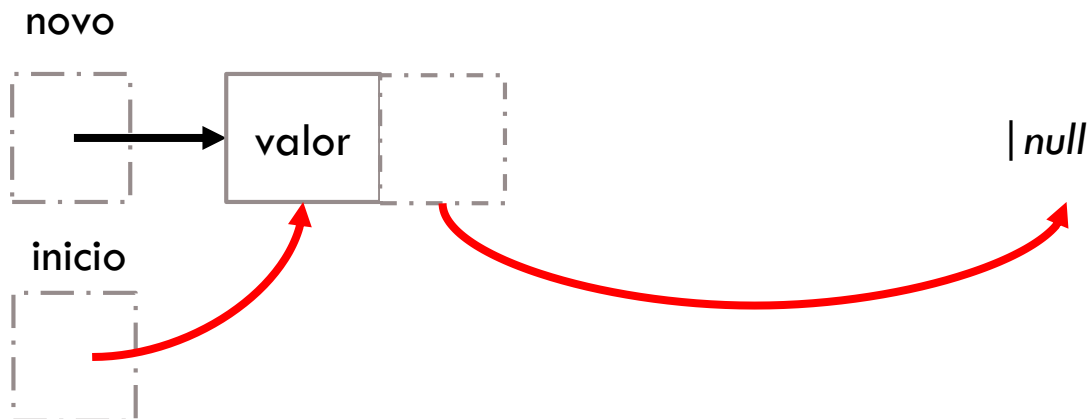


# Inserção (no início da lista)

29

- Início aponta para elemento apontado por novo

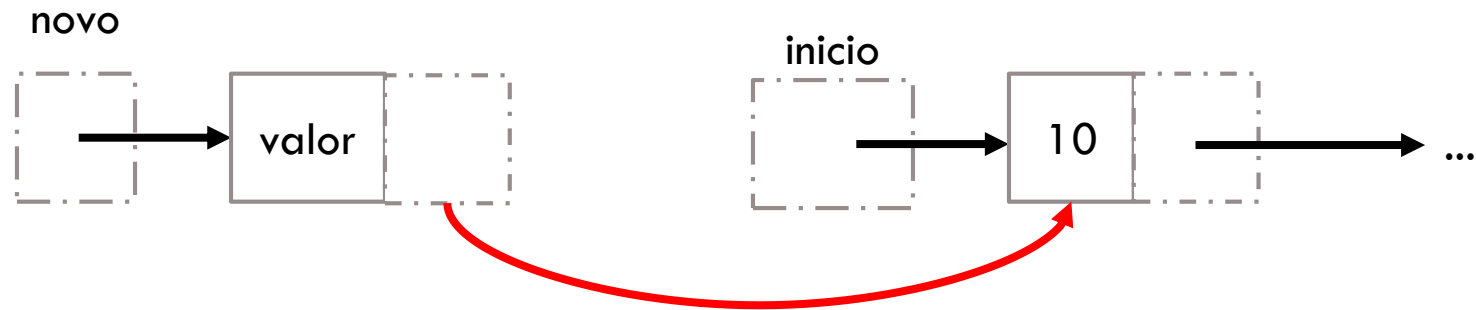
```
void insercao_inicio_lista(Lista *p_l, int valor){  
    No *novo = malloc(sizeof(No));  
    novo->chave = valor;  
    novo->prox = p_l->inicio;  
    p_l->inicio = novo;  
}
```



*p\_l* aponta para o  
registro de lista

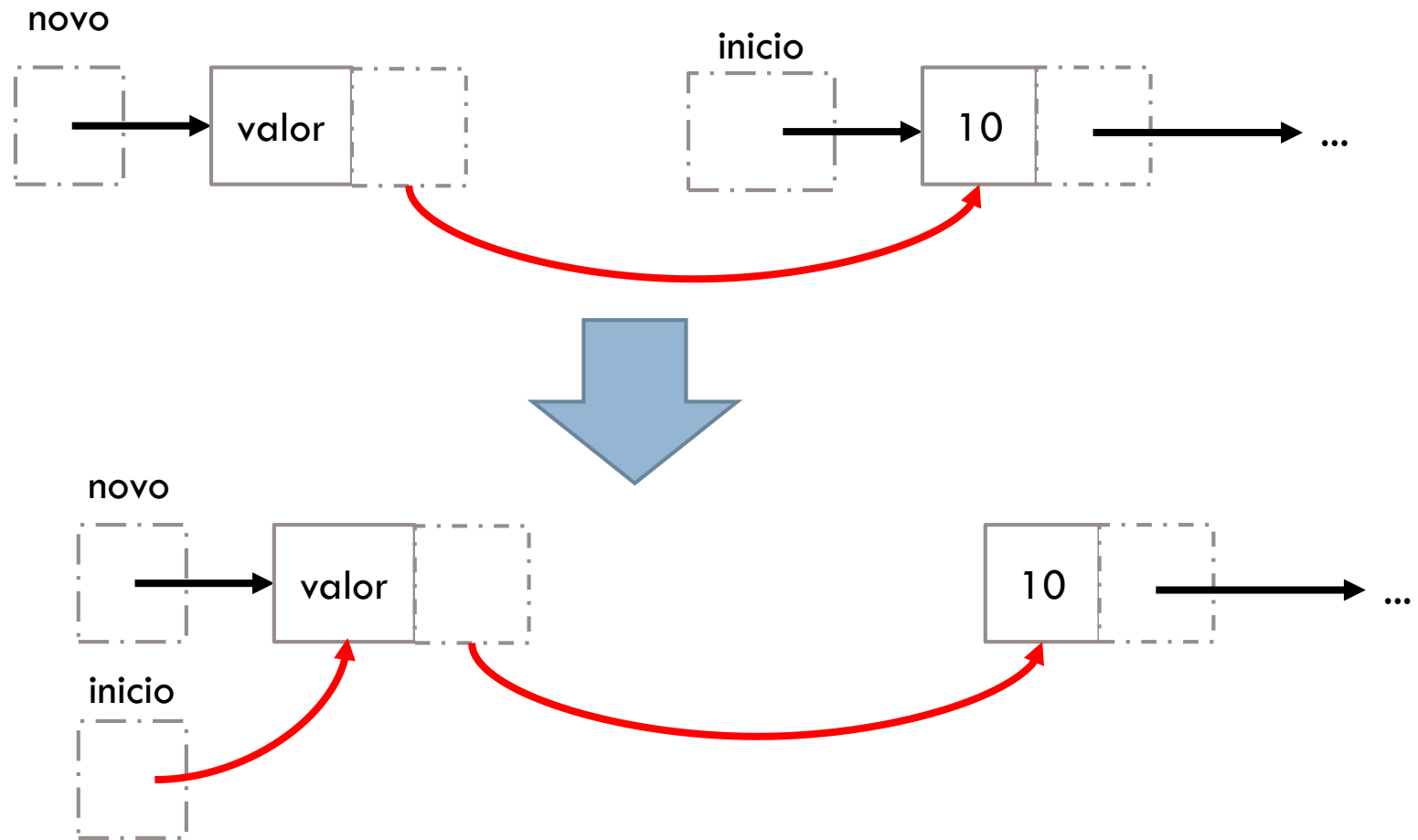
# Inserção (no início da lista NÃO VAZIA)

30



# Inserção (no início da lista NÃO VAZIA)

31



# Inserção (código refinado)

32

- Verificar o resultado de *malloc*

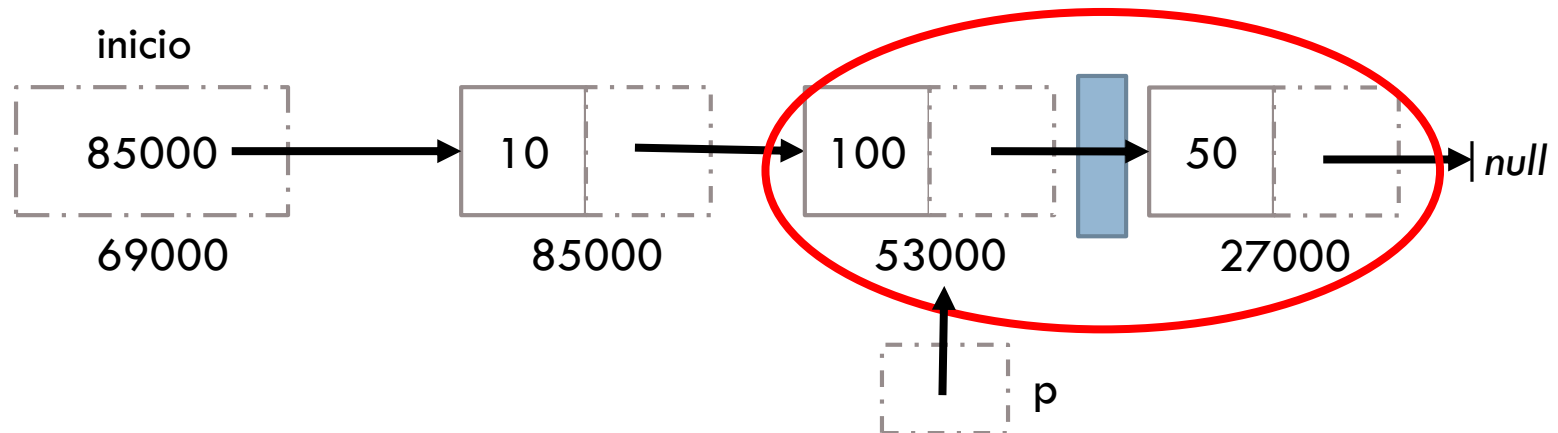
```
bool insercao_inicio_lista(Lista *p_l, int valor){  
    No *novo = malloc(sizeof(No));  
    if (novo==NULL) //falta de memória  
        return false;  
    novo->chave = valor;  
    novo->prox = p_l->inicio;  
    p_l->inicio = novo;  
    return true;  
}
```



# Inserção (no meio da lista)

33

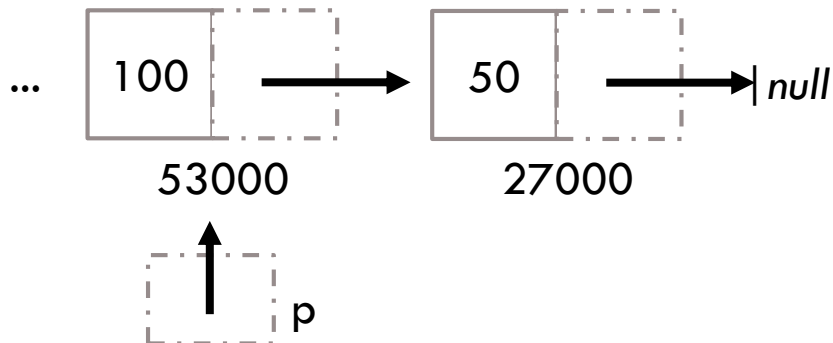
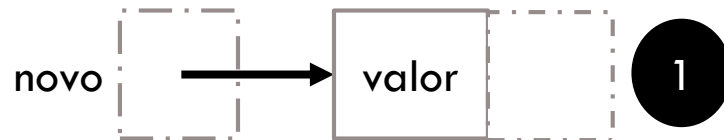
- Entre um nó apontado por  $p$  e o seguinte



# Passo-a-passo inserção no meio da lista

34

□ Aloque um novo nó 1

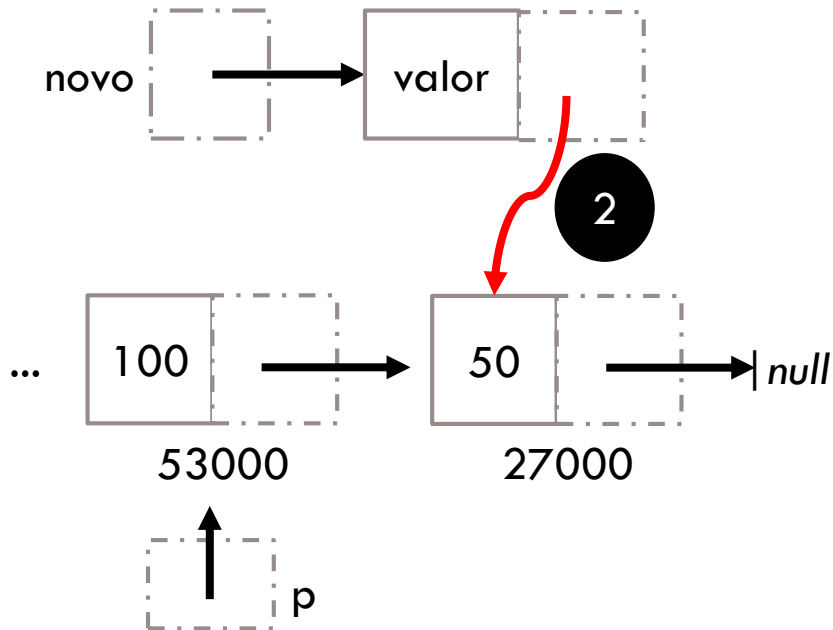


# Passo-a-passo inserção no meio da lista

35

□ Aloque um novo nó 1

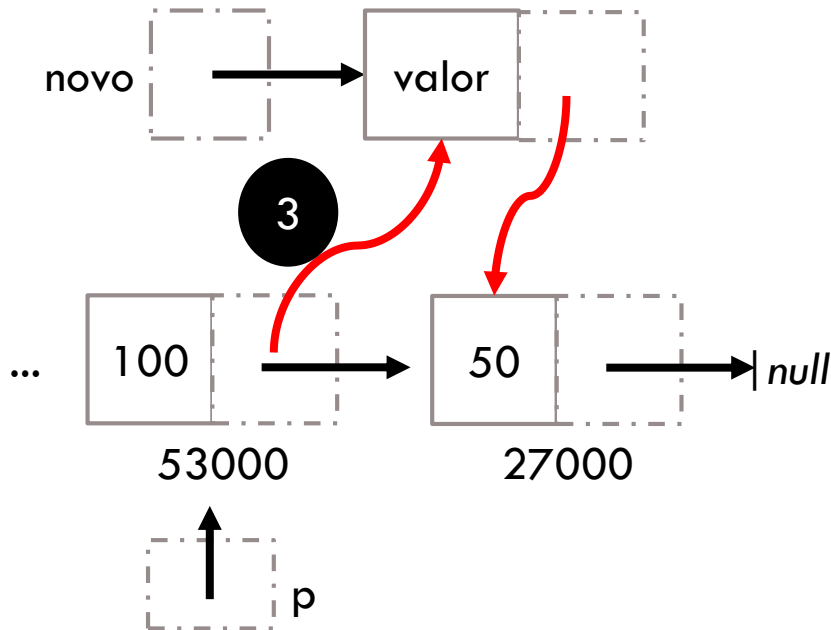
□ Campo *prox* de *novo* aponta para o *prox* de *p* 2



# Passo-a-passo inserção no meio da lista

36

- Aloque um novo nó 1
- Campo *prox* de *novo* aponta o *prox* de *p* 2
- Campo *prox* de *p* aponta para o novo 3



# Operação de inserção no meio da lista

37

```
bool insercao_meio_lista(No *p, int valor){  
    No *novo = (No*) malloc(sizeof(No)); 1  
    if (novo==NULL)  
        return false;  
    novo->chave = valor;  
    novo->prox = p->prox; 2  
    p->prox = novo; 3  
    return true;  
}
```

# Operação de inserção no meio da lista

38

```
bool insercao_meio_lista(No *p, int valor){  
    No *novo = (No*) malloc(sizeof(No)); 1  
    if (novo==NULL)  
        return false;  
    novo->chave = valor;  
    novo->prox = p->prox; 2  
    p->prox = novo; 3  
    return true;  
}
```

*Por quê este código não funciona para inserção no início?*

# Operação de Inserção no meio da lista

39

```
bool insercao_meio_lista(No *p, int valor){  
    No *novo = (No*) malloc(sizeof(No)); 1  
    if (novo==NULL)  
        return false;  
    novo->chave = valor;  
    novo->prox = p->prox; 2  
    p->prox = novo; 3  
    return true;  
}
```

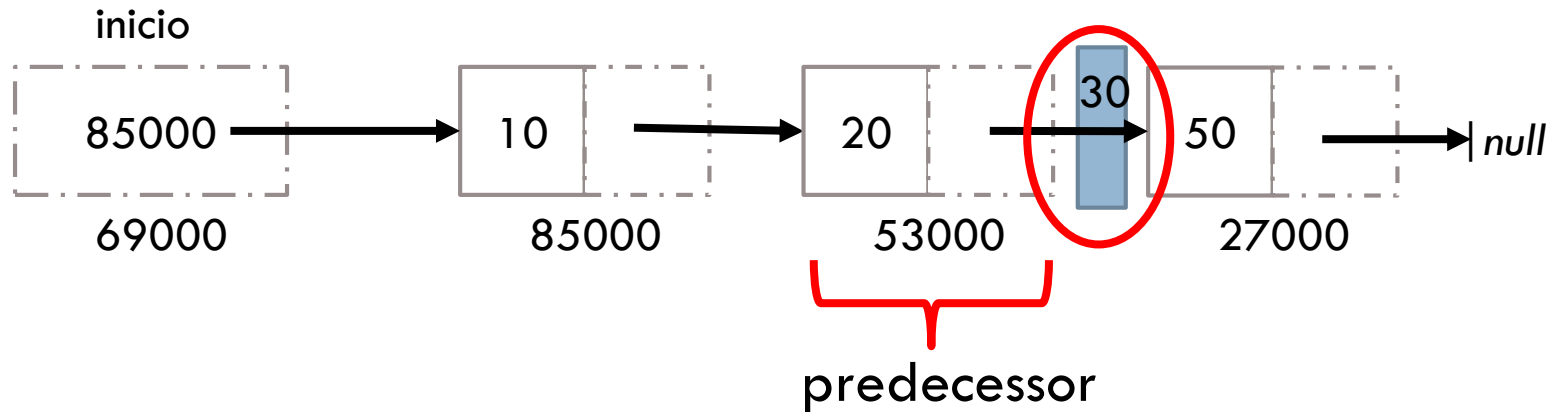
*Por quê este código não funciona para inserção no início?*

*Como a função externa conhece o ponteiro para o início da lista?*

# Inserção (lista ordenada)

40

□ Exemplo: Inserir o elemento 30





# Dificuldades de inserção na lista ordenada

41

- Necessário conhecer o antecessor do elemento que se quer inserir
- Não se tem o ponteiro  $p$  antecessor como argumento
  - ▣ Argumento é o ponteiro inicio da lista

# Dificuldades de inserção na lista ordenada

42

- Necessário conhecer o antecessor do elemento que se quer inserir
- Não se tem o ponteiro  $p$  antecessor como argumento
  - ▣ Argumento é o ponteiro início da lista
- Precisa-se procurar onde se inserir o novo elemento na lista
  - ▣ Depois que encontrar, pode-se aplicar o procedimento de inserção no meio

# Dificuldades de inserção na lista ordenada

43

- ❑ Necessário conhecer o antecessor do elemento que se quer inserir
- ❑ Não se tem o ponteiro  $p$  antecessor como argumento
  - ❑ Argumento é o ponteiro inicio da lista
- ❑ Precisa-se procurar onde se inserir o novo elemento na lista
  - ❑ Depois que encontrar, pode-se aplicar o procedimento de inserção no meio
- ❑ Não permitir a inserção de elementos repetidos

# Procedimento auxiliar de busca

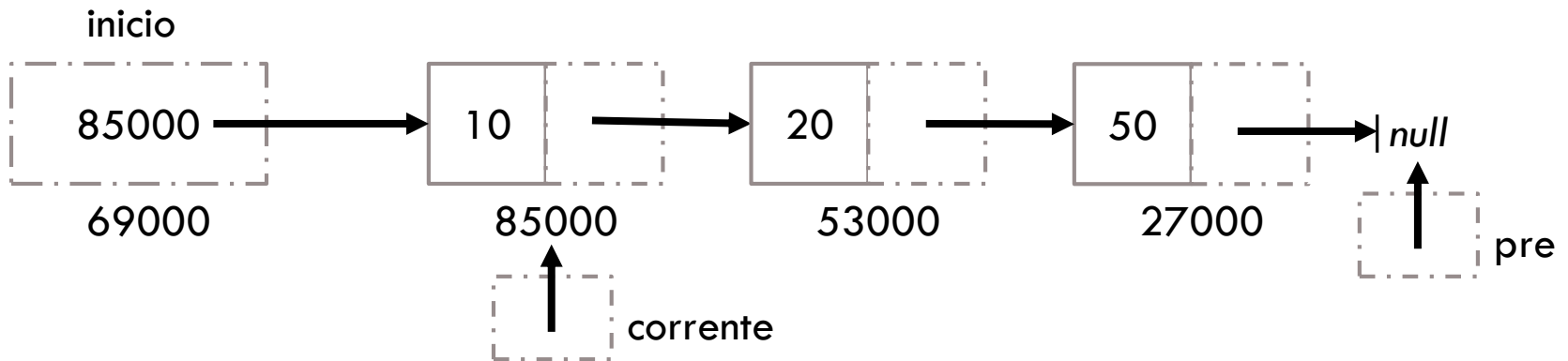
44

- Deve retornar:
  - ▣ Endereço do elemento buscado, se ele existir
  - ▣ Endereço do elemento predecessor do buscado
    - Elemento buscado existindo ou não na lista

# Elemento corrente e predecessor

45

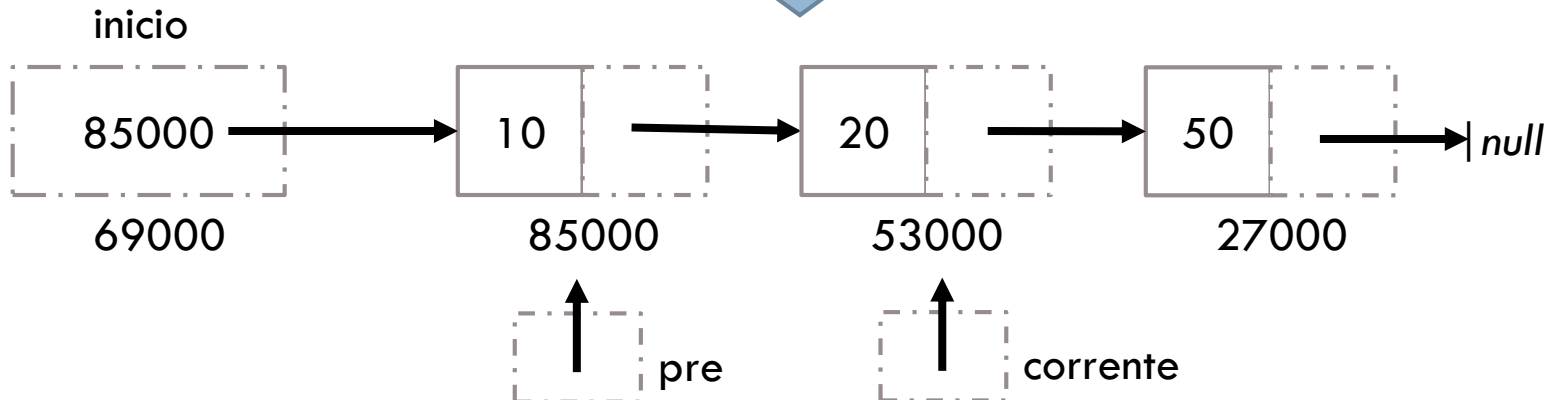
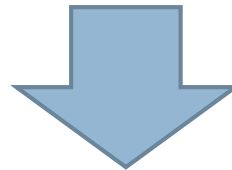
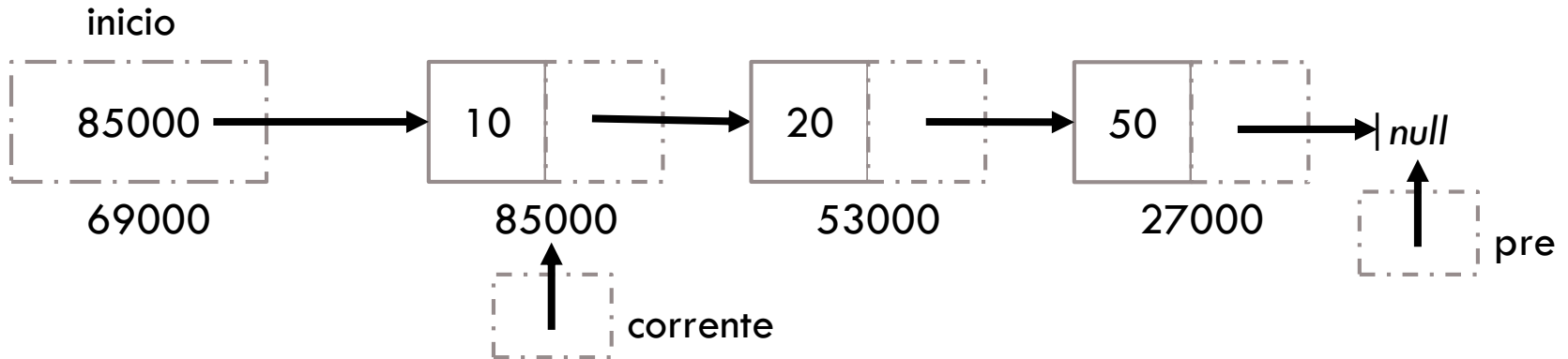
Busca do elemento = 30



# Elemento corrente e predecessor

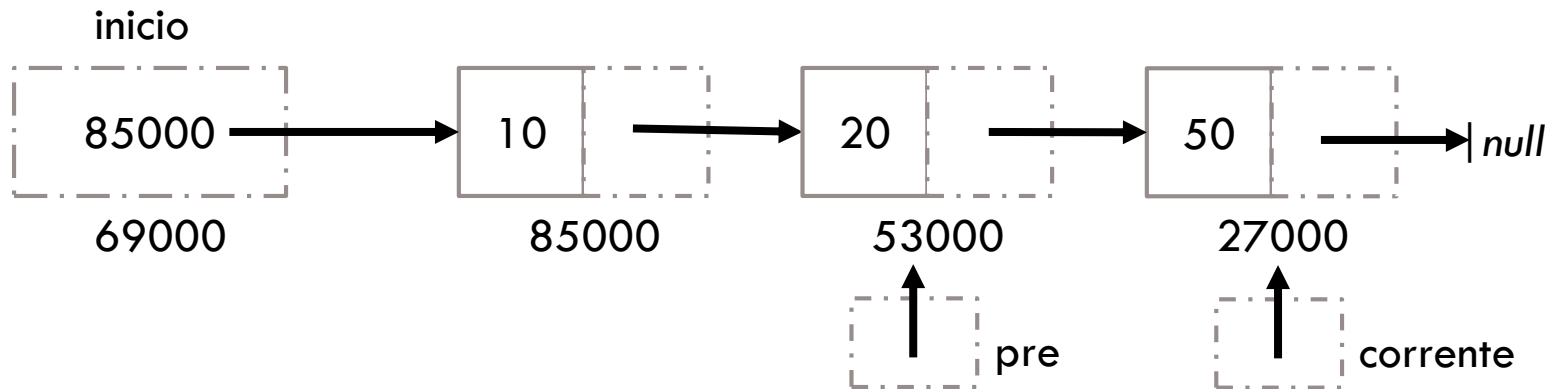
46

Busca do elemento = 30



# Elemento corrente e predecessor

47 Busca do elemento = 30



- Elemento corrente (50) é maior que a chave de busca (30)
  - ▣ O algoritmo para neste estágio
- Ponteiro *pre* armazena endereço para o último elemento menor que a chave de busca

# Procedimento auxiliar de busca

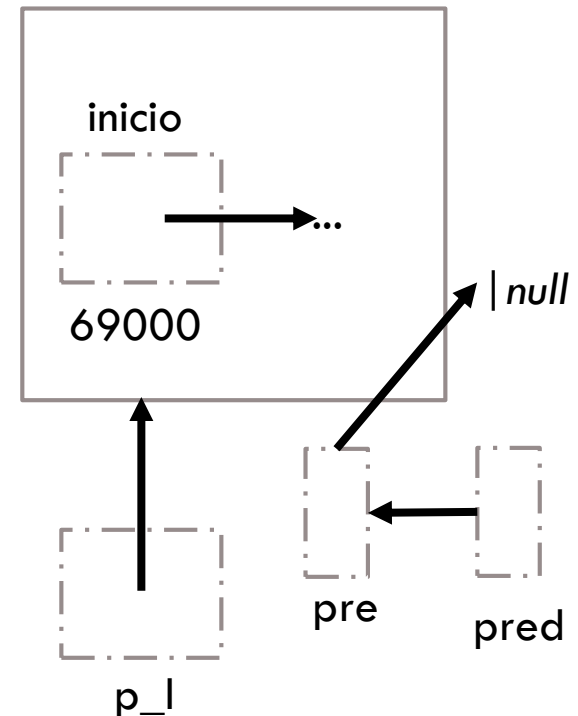
48

```
No* buscar_lista(Lista *p_l, int ch_busca, No **pred){  
    (...)  
}
```

## Código cliente

```
Lista l;  
int chave_busca=30;  
No *pre = NULL;  
buscar_lista(&l, chave_busca, &pre);
```

Registro Lista l





# Procedimento auxiliar de busca

49

```
No* buscar_lista(Lista *p_l, int ch_busca, No** pred){  
    *pred = NULL;  
    No* corrente = p_l->inicio;
```

Nó corrente aponta  
para o início da lista

```
}
```

# Procedimento auxiliar de busca

50

```
No* buscar_lista(Lista *p_l, int ch_busca, No** pred){  
    *pred = NULL;  
    No* corrente = p_l->inicio;  
  
    while ((corrente != NULL) && (corrente->chave < ch_busca)) {  
        *pred = corrente;  
        corrente = corrente->prox;  
    }  
  
}
```

Percorre a lista enquanto  
não chegar no final dela  
ou encontrar uma chave  
de valor maior

# Procedimento auxiliar de busca

51

```
No* buscar_lista(Lista *p_l, int ch_busca, No** pred){
    *pred = NULL;
    No* corrente = p_l->inicio;

    while ((corrente != NULL) && (corrente->chave < ch_busca)) {
        *pred = corrente;
        corrente = corrente->prox;
    }

    if ((corrente != NULL) && (corrente->chave == ch_busca))
        return corrente;

    return NULL;
}
```

Se encontrou o elemento,  
retorna o mesmo

# Procedimento auxiliar de busca

52

```
No* buscar_lista(Lista *p_l, int ch_busca, No** pred){
    *pred = NULL;
    No* corrente = p_l->inicio;

    while ((corrente != NULL) && (corrente->chave < ch_busca)) {
        *pred = corrente;
        corrente = corrente->prox;
    }

    if ((corrente != NULL) && (corrente->chave == ch_busca))
        return corrente;

    return NULL;
}
```

*pred* apontará para o último elemento menor que a chave de busca

# Procedimento de inserção em lista ordenada

53

```
bool inserir_lista_ord(Lista *p_l, int valor){
    No* q, pre;
    pre=NULL;
    q = buscar_lista(p_l, valor, &pre);

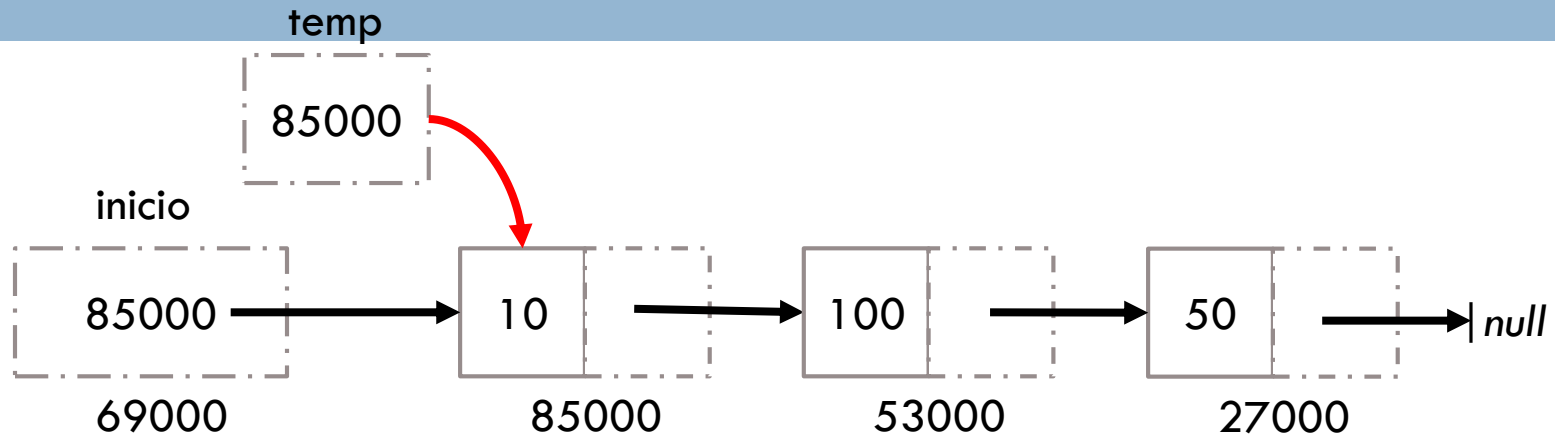
    if (q!=NULL) // elemento já existe na lista
        return false;

    if (pre==NULL) // lista vazia (inserção no inicio)
        return insercao_inicio_lista(p_l, valor);
    else // (inserção no meio)
        return insercao_meio_lista(pre, valor);
}
```

## Algoritmos de remoção em lista ligada

# Remoção (no início da lista)

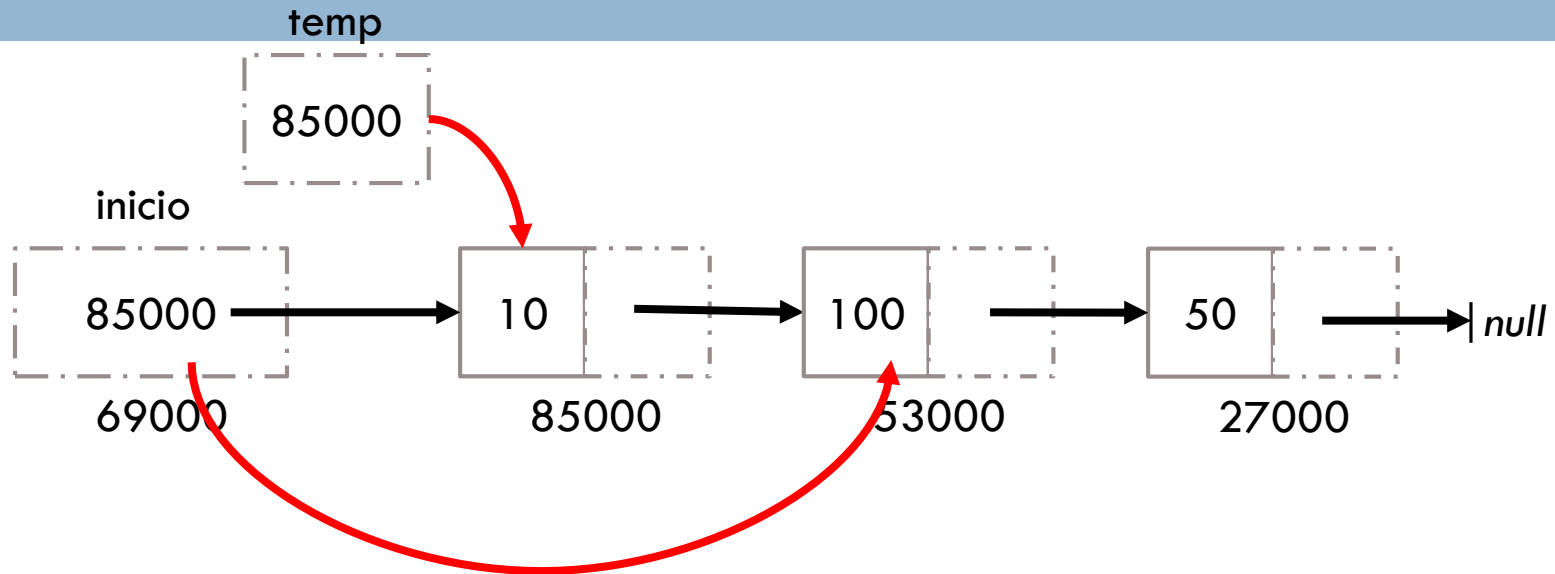
55



```
bool remover_inicio_lista(Lista *p_l){  
    No* temp = p_l->inicio;  
  
}
```

# Remoção (no início da lista)

56

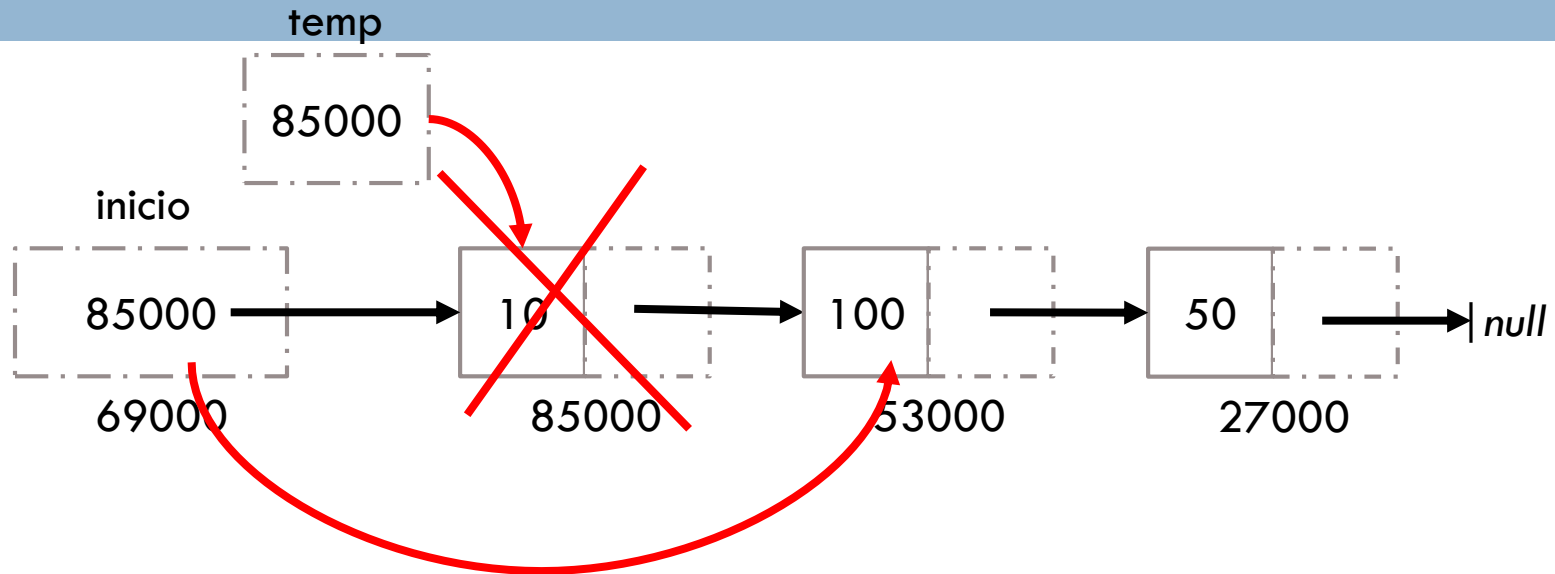


```
bool remover_inicio_lista(Lista *p_l){  
    No* temp = p_l->inicio;  
    p_l->inicio = temp->prox;  
}
```



# Remoção (no início da lista)

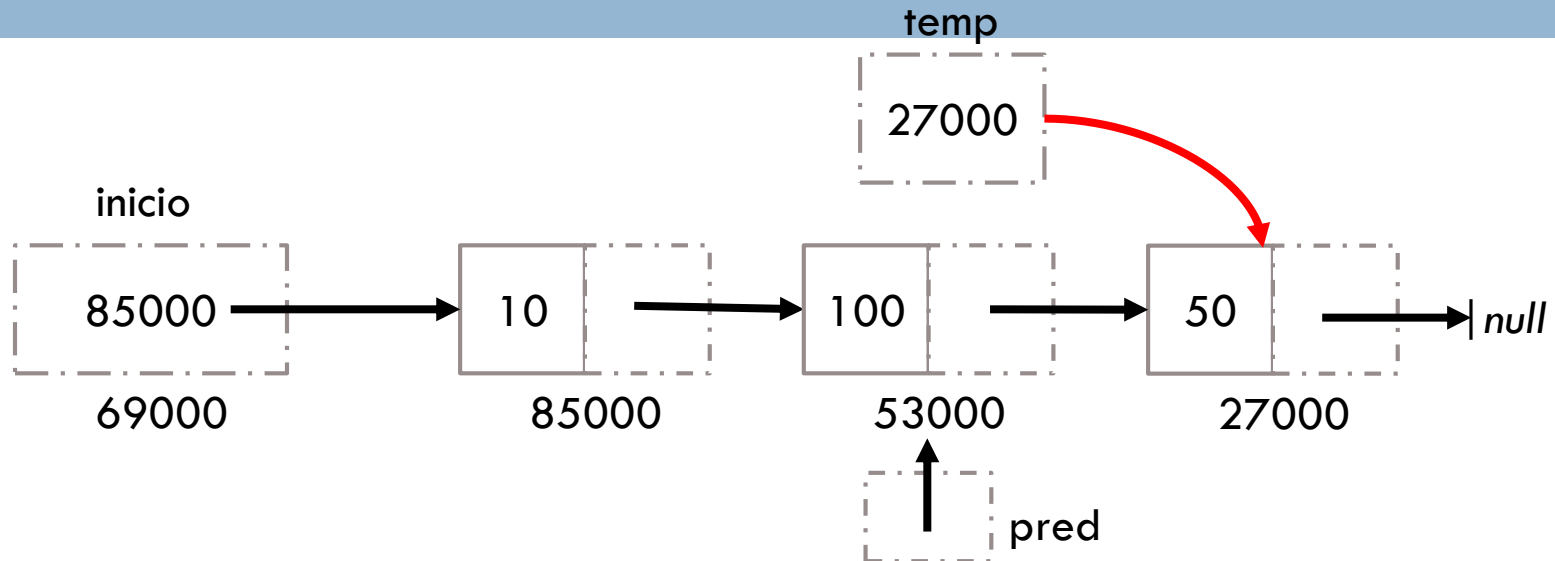
57



```
bool remover_inicio_lista(Lista *p_l){  
    No* temp = p_l->inicio;  
    p_l->inicio = temp->prox;  
    free(temp);  
    return true;  
}
```

# Remoção (no meio da lista)

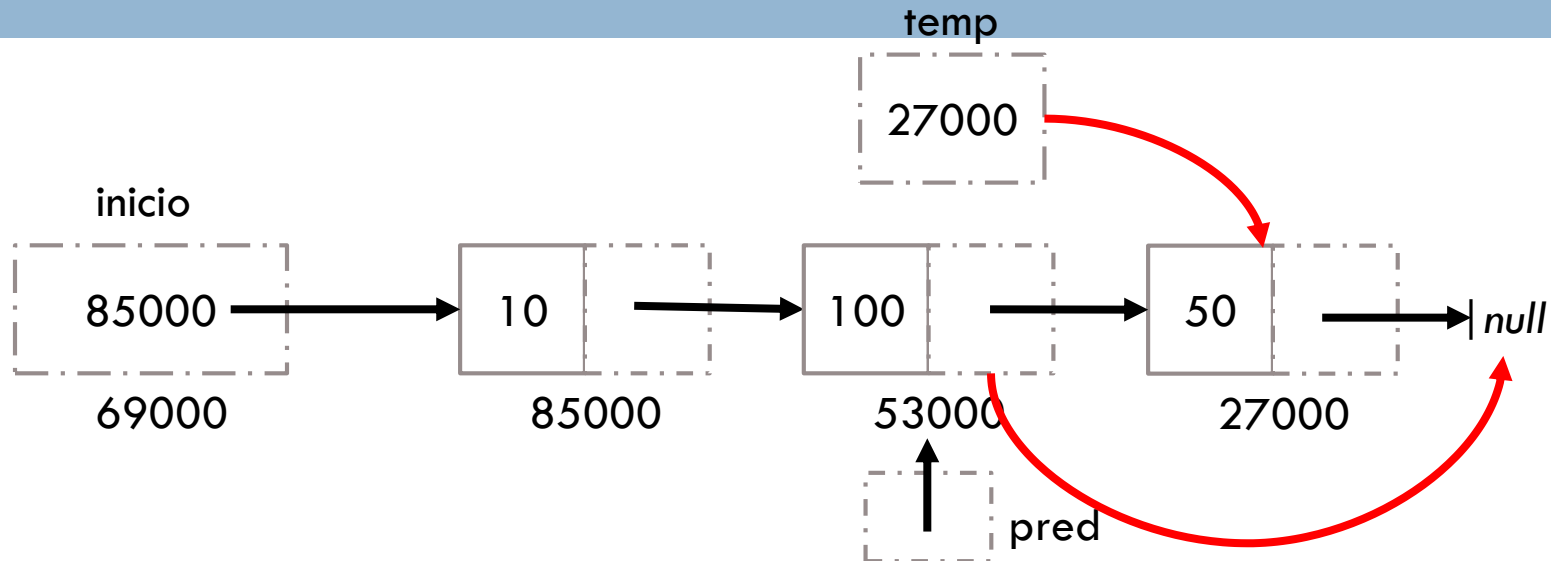
58



```
bool remover_meio_lista(No* pred){  
    No* temp = pred->prox;  
  
}
```

# Remoção (no meio da lista)

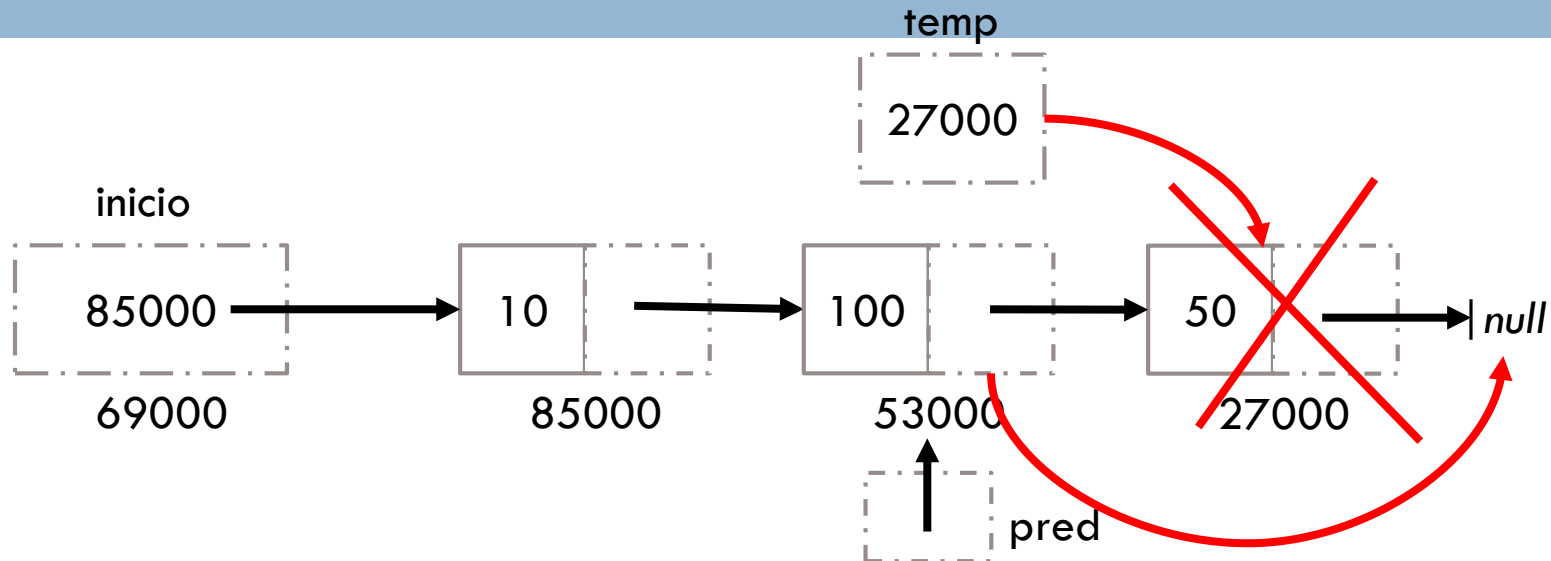
59



```
bool remover_meio_lista(No* pred){  
    No* temp = pred->prox;  
    pred->prox = temp->prox;  
}
```

# Remoção (no meio da lista)

60



```
bool remover_meio_lista(No* pred){  
    No* temp = pred->prox;  
    pred->prox = temp->prox;  
    free(temp);  
    return true;  
}
```

# Remoção (chave indicada pelo usuário)

61

- Buscar elemento com a chave na lista

# Remoção (chave indicada pelo usuário)

62

- Buscar elemento com a chave na lista
- Se houver o elemento, exclui-lo através do acerto dos ponteiros envolvidos
  - ▣ Retornar *true*

# Remoção (chave indicada pelo usuário)

63

- Buscar elemento com a chave na lista
- Se houver o elemento, exclui-lo através do acerto dos ponteiros envolvidos
  - ▣ Retornar *true*
- É necessário conhecer o elemento predecessor do nó que será excluído

# Remoção (chave indicada pelo usuário)

64

```
bool remover_lista(Lista *p_l, int chave_usu){  
    No* q, pred;  
    pred=NULL;  
    q = buscar_lista(p_l, chave_usu, &pred);  
  
}
```



# Remoção (chave indicada pelo usuário)

65

```
bool remover_lista(Lista *p_l, int chave_usu){  
    No* q, pred;  
    pred=NULL;  
    q = buscar_lista(p_l, chave_usu, &pred);  
  
    if (q==NULL) // elemento não existe na lista  
        return false;  
  
}
```

# Remoção (chave indicada pelo usuário)

66

```
bool remover_lista(Lista *p_l, int chave_usu){
    No* q, pred;
    pred=NULL;
    q = buscar_lista(p_l, chave_usu, &pred);

    if (q==NULL) // elemento não existe na lista
        return false;

    if (pred==NULL) // remover no inicio da lista
        return remover_inicio_lista(p_l);
    else // remover no meio da lista
        return remover_meio_lista(pred);
}
```

# Impressão da lista

67

- Itera na lista até encontrar *NULL*
  - ▣ A cada iteração imprime o valor da chave

```
void imprimir_lista(Lista *p_l){  
    No* temp = p_l->inicio;  
    while(temp!=NULL){  
        println("%d", temp->valor);  
        temp=temp->prox; //aponta para o próximo nó  
    }  
}
```

# Destruição da lista

68

- Itera na lista até encontrar *NULL*
  - ▣ A cada iteração desaloca a memória do nó

```
void destruir_lista(Lista *p_l){  
    No* temp = p_l->inicio;  
    while(temp!=NULL){  
        No* prox = temp->prox;  
        free(temp);  
        temp=prox;  
    }  
    p_l->inicio=null;  
}
```

# Síntese

69

- Estudamos a definição e a modelagem de estruturas ligadas
  - ▣ Requerem um ponteiro para o elemento subsequente

# Síntese

70

- Estudamos a definição e a modelagem de estruturas ligadas
  - ▣ Requerem um ponteiro para o elemento subsequente
- Operações de inicialização, inserção, remoção, impressão e liberação de nós

# Síntese

71

- Estudamos a definição e a modelagem de estruturas ligadas
  - ▣ Requerem um ponteiro para o elemento subsequente
- Operações de inicialização, inserção, remoção, impressão e liberação de nós
- Listas ligadas permitem alocação dos elementos da lista conforme o necessário
  - ▣ Melhor economia e gerenciamento da estrutura

# Exercício

72

- Defina um procedimento para remover o  $k$ -ésimo elemento de uma lista ligada