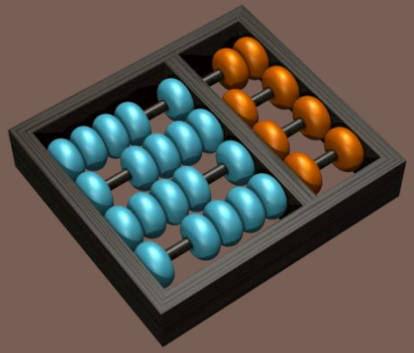


UNICAMP

VARIAÇÕES DE LISTAS LIGADAS

Prof.: Julio Cesar dos Reis

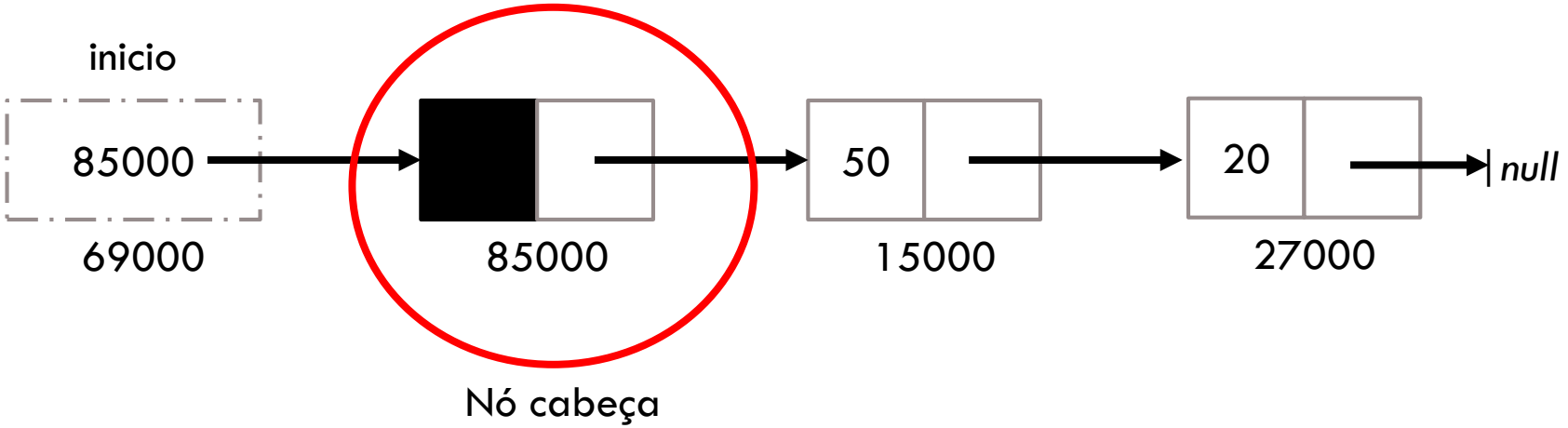
Roteiro

- Variações de lista ligada
- Noções de eficiência de algoritmos
- Comparação de arranjos com listas ligadas

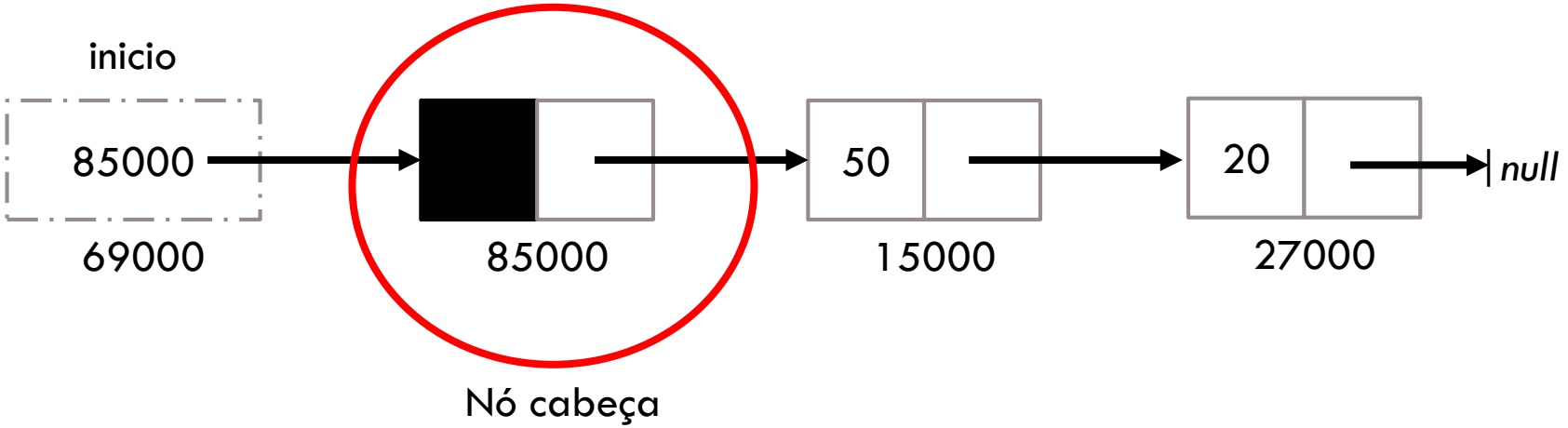
Variações de lista ligada

- Lista com cabeça
- Lista circular
- Lista circular com cabeça
- Lista duplamente ligada
- Lista duplamente ligada circular
- Lista duplamente ligada circular com cabeça

Listas com cabeça



Listas com cabeça



- Características
  - Ponteiro inicial sempre referencia o nó cabeça
  - Procedimentos idênticos para adicionar no início ou no meio da lista
  - Nó cabeça deve ser ignorado ao percorrer a lista
  - Pode ser útil para guardar informações adicionais

Estrutura da lista

Elemento nó

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

Lista ligada

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

Inicialização da lista com cabeça

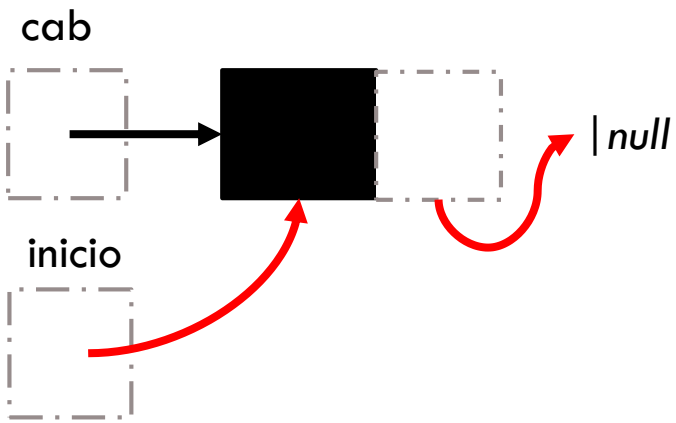
- Necessário criar um nó cabeça
- Ponteiro inicial aponta para o nó cabeça
- Campo próximo do nó cabeça aponta para *null*

Inicialização da lista com cabeça


```
No* iniciar_lista_cab(Lista *p_l){
    No *cab = malloc(sizeof(No));
    p_l->inicio = cab;
   cab->prox = null;
   return cab;
}
```

Código cliente

```
Lista l;
iniciar_lista_cab(&l);
```

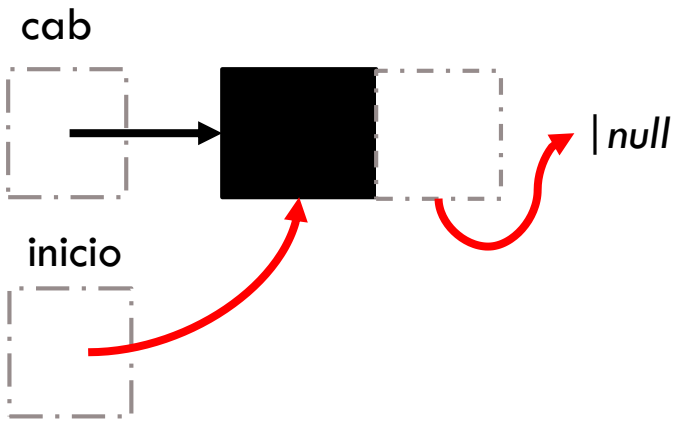


Inicialização da lista com cabeça


```
No* iniciar_lista_cab(Lista *p_l){
    No *cab = malloc(sizeof(No));
    p_l->inicio = cab;
   cab->prox = null;
   return cab;
}
```

Código cliente

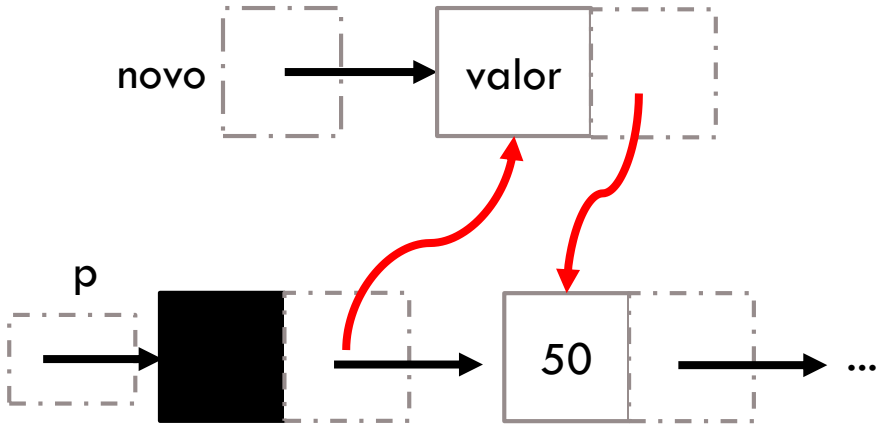
```
Lista l;
iniciar_lista_cab(&l);
```



Como verificar se a lista está vazia?

Inserção na lista com cabeça

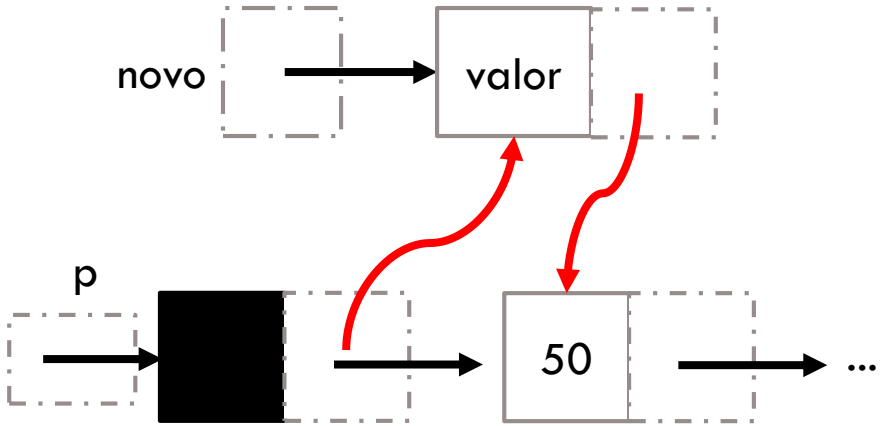

```
bool insercao_lista_cab(No *p, int valor){
   No *novo = (No*) malloc(sizeof(No));
   if (novo==NULL)  return false;
   novo->chave = valor;
   novo->prox = p->prox;
   p->prox = novo;
   return true;
}
```



Inserção na lista com cabeça


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

Ponteiro para Nó;
Não é ponteiro de inicio da lista
(p_l->inicio->prox)



Exemplo com lista

- Leia *n* números pares e os insira em uma lista

```
#include <stdio.h>
#include <stdlib.h>
int main (){
    Lista l;
    No* cab = iniciar_lista_cab(&l);
    int n_par;
    do {
       printf("Digite um número par: "); scanf("%d", &n_par);
       if (n_par%2==0)
            insercao_lista_cab(cab, n_par);
     } while (n_par%2==0);
     imprimir_lista(&l);
     destruir_lista(&l);
}
```

Exemplo com lista

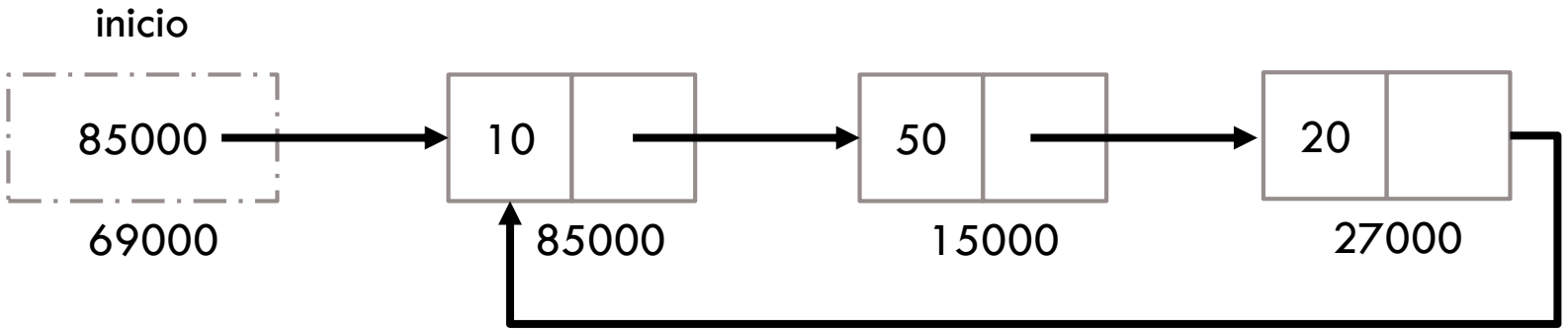
- Leia *n* números pares e os insira em uma lista

```
#include <stdio.h>
#include <stdlib.h>
int main (){
    Lista l;
    No* cab = iniciar_lista_cab(&l);
    int n_par;
    do {
       printf("Digite um número par: "); scanf("%d", &n_par);
       if (n_par%2==0)
            insercao_lista_cab(cab, n_par);
     } while (n_par%2==0);
     imprimir_lista(&l);
     destruir_lista(&l);
}
```

Procedimento de impressão da lista com cabeça é diferente?

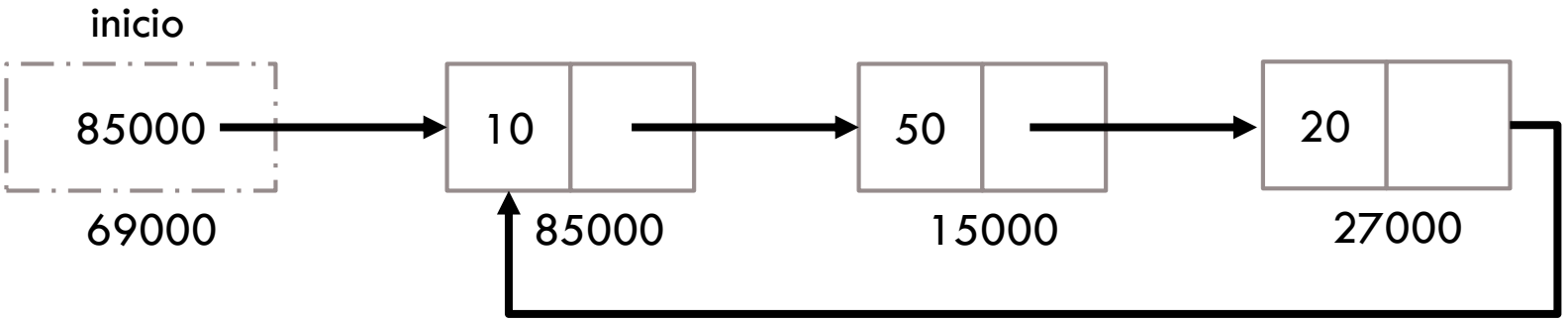
Lista circular

- Último nó aponta para o primeiro



Lista circular

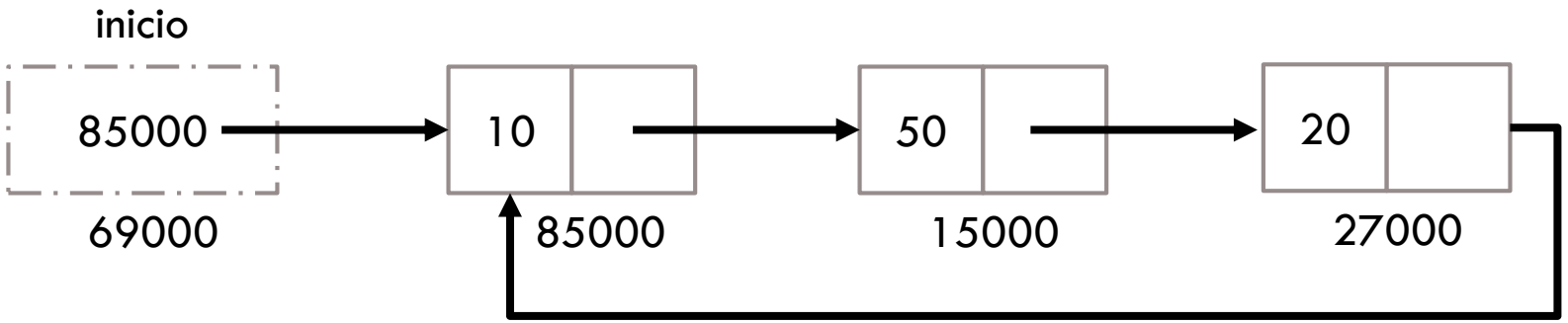
- Último nó aponta para o primeiro



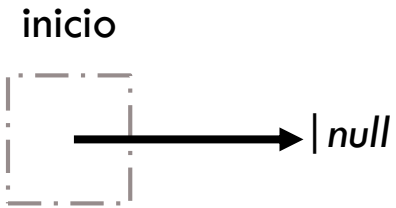
Como verificar se a lista está vazia?

Lista circular

- Último nó aponta para o primeiro



Como verificar se a lista está vazia?

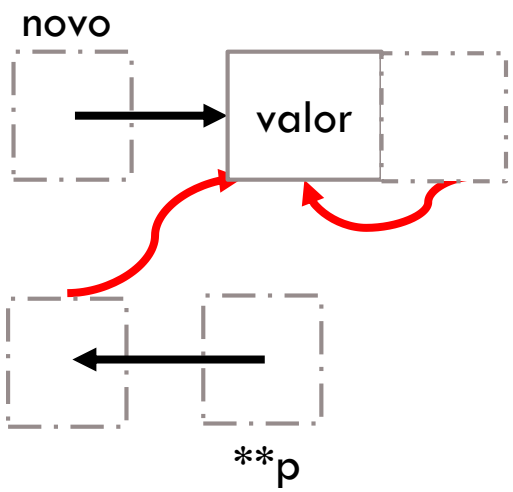


Inserção em lista circular


```
bool insercao_lista_circ(No **p, int valor){
   No *novo = (No*) malloc(sizeof(No));
   if (novo==NULL)  return false;
   novo->chave = valor;









}
```

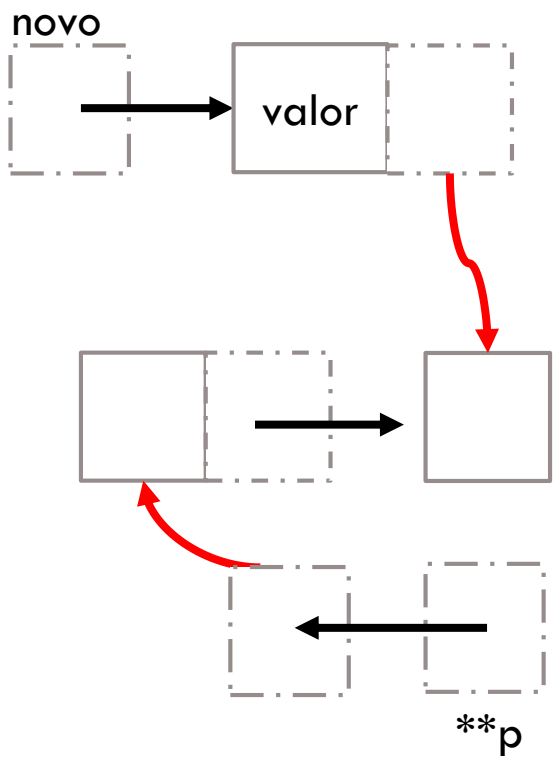
Inserção em lista circular


```
bool insercao_lista_circ(No **p, int valor){
   No *novo = (No*) malloc(sizeof(No));
   if (novo==NULL)  return false;
   novo->chave = valor;
   if ((*p) == NULL) { //lista vazia
       (*p) = novo;
       novo->prox = novo;
    }




}
```



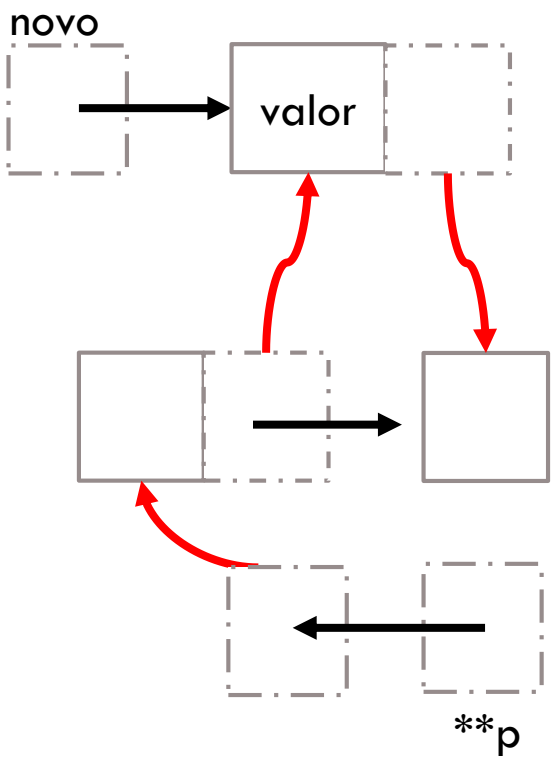
Inserção em lista circular


```
bool insercao_lista_circ(No **p, int valor){
   No *novo = (No*) malloc(sizeof(No));
   if (novo==NULL)  return false;
   novo->chave = valor;
   if ((*p) == NULL) { //lista vazia
       (*p) = novo;
       novo->prox = novo;
    } else {
       novo->prox = (*p)->prox;
      (*p)->prox = novo;
    }
  return true;
}
```



Inserção em lista circular


```
bool insercao_lista_circ(No **p, int valor){
   No *novo = (No*) malloc(sizeof(No));
   if (novo==NULL)  return false;
   novo->chave = valor;
   if ((*p) == NULL) { //lista vazia
       (*p) = novo;
       novo->prox = novo;
    } else {
       novo->prox = (*p)->prox;
       (*p)->prox = novo;
    }
  return true;
}
```



Inserção em lista circular


```
bool insercao_lista_circ(No **p, int valor){
   No *novo = (No*) malloc(sizeof(No));
   if (novo==NULL)  return false;
   novo->chave = valor;
   if ((*p) == NULL) { //lista vazia
       (*p) = novo;
       novo->prox = novo;
    } else {
       novo->prox = (*p)->prox;
      (*p)->prox = novo;
    }
  return true;
}
```

Como modificar este procedimento para receber como parâmetro um ponteiro para Lista?

Inserção em lista circular


```
bool insercao_lista_circ(Lista * p_l, int valor){
   No *novo = (No*) malloc(sizeof(No));
   if (novo==NULL)  return false;
   novo->chave = valor;
   if (p_l ->inicio == NULL) { //lista vazia
       p_l ->inicio = novo;
       novo->prox = novo;
    } else {
       novo->prox = p_l->inicio;
       p_l->inicio= novo;
    }
  return true;
}
```

Isso funciona?
Por quê não?

Remoção em lista circular


```
bool remover_lista_circ(Lista *p_l, No *no_re) {
      if (no_re->prox == no_re) {// se contém apenas um elemento
           p_l->inicio = NULL;
       }









}
```

Remoção em lista circular


```
bool remover_lista_circ(Lista *p_l, No *no_re) {
      if (no_re->prox == no_re) {// se contém apenas um elemento
           p_l->inicio = NULL;
       } else {// se nó removido é o início da lista, avança a lista
          if (p_l->inicio == no_re)
               p_l->inicio = p_l->inicio->prox;









}
```

Remoção em lista circular


```
bool remover_lista_circ(Lista *p_l, No *no_re) {
      if (no_re->prox == no_re) {// se contém apenas um elemento
           p_l->inicio = NULL;
       } else {// se nó removido é o início da lista, avança a lista
          if (p_l->inicio == no_re)
               p_l->inicio = p_l->inicio->prox;
          // encontra Nó anterior percorrendo a lista circular
          No *ant = no_re->prox;
          while (ant->prox != no_re)
              ant = ant->prox;
          // remove nó da lista
          ant->prox = no_re->prox;
       }   free(no_re); return true;
}
```

Percorrendo uma lista circular

- Verificar se chegou ao fim

```
void imprimir_lista_circ(Lista *p_l) {
        No *aux = p_l->inicio;
        do {
                printf("%d ", aux->chave);
                aux = aux->prox;
        } while (aux != p_l->inicio);
}
```

Percorrendo uma lista circular

- Verificar se chegou ao fim

```
void imprimir_lista_circ(Lista *p_l) {
        No *aux = p_l->inicio;
        do {
                printf("%d ", aux->chave);
                aux = aux->prox;
        } while (aux != p_l->inicio);
}
```

- Procedimento funciona para lista vazia?

Percorrendo uma lista circular

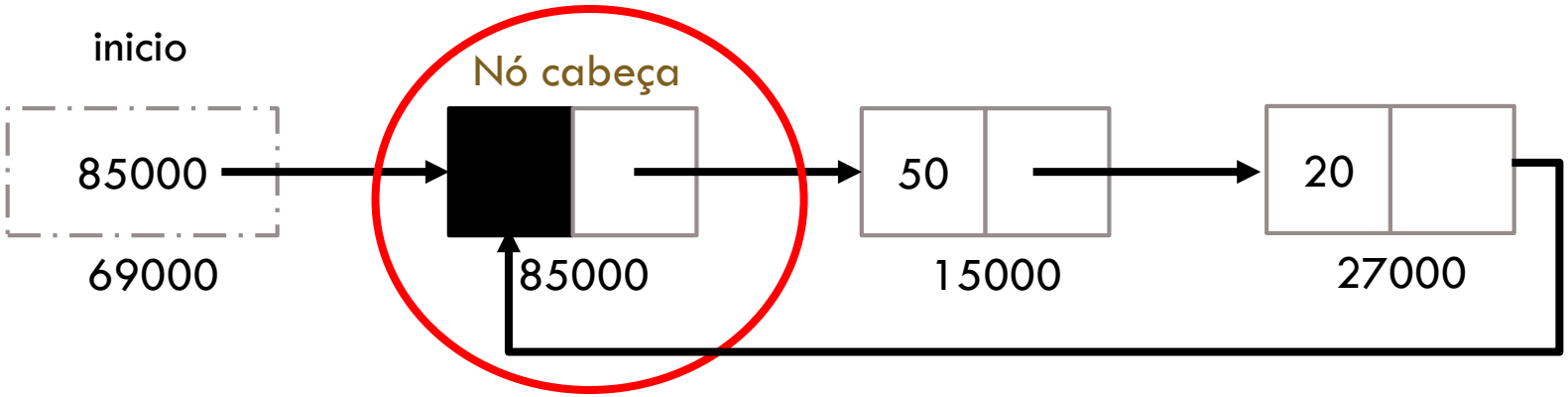
- Verificar se chegou ao fim

```
void imprimir_lista_circ(Lista *p_l) {
        No *aux = p_l->inicio;
        do {
                printf("%d ", aux->chave);
                aux = aux->prox;
        } while (aux != p_l->inicio);
}
```

- Procedimento funciona para lista vazia?
- Qual a diferença se usar a estrutura while {...} em vez de do {...} while?

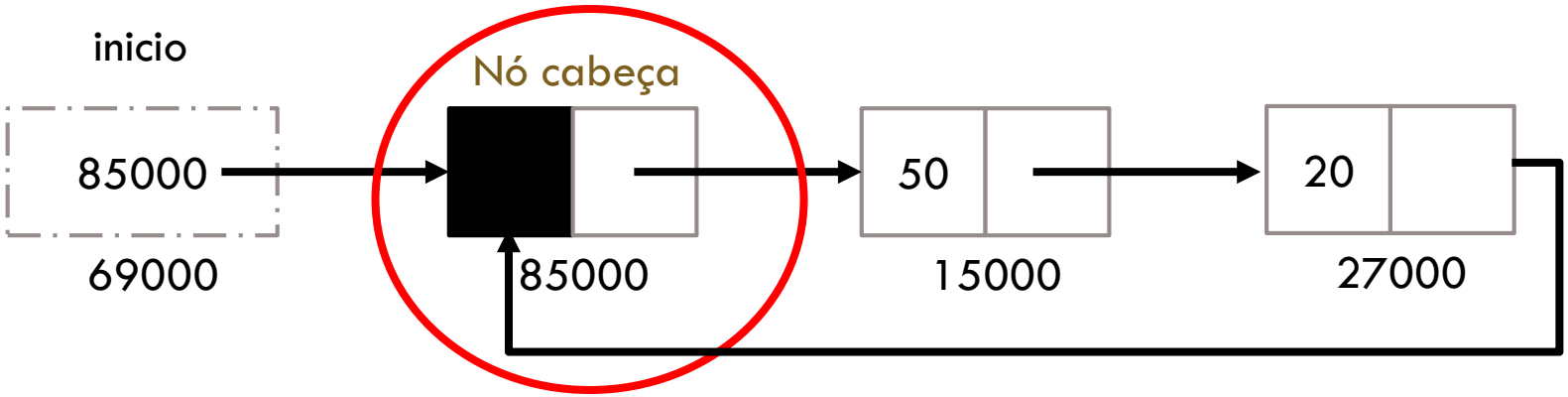
Lista circular com cabeça

- Último nó aponta para o nó cabeça

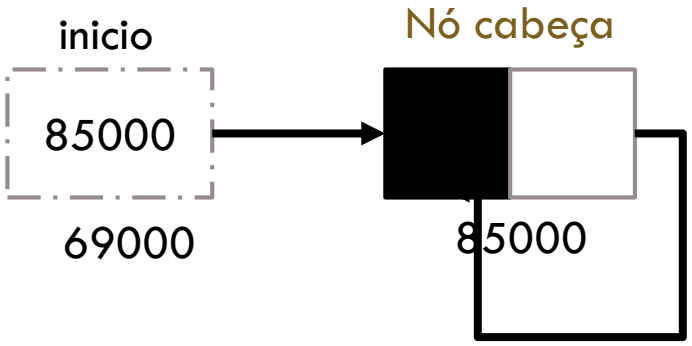


Lista circular com cabeça

- Último nó aponta para o nó cabeça



- Lista vazia



Operações na lista circular com cabeça

- Inicialização
- Comprimento da lista
- Imprimir a lista
- Busca de elemento
- Inserção
- Remoção

Inicialização

- Criar um nó cabeça
- O início da lista aponta para o nó cabeça
- O nó cabeça aponta para ele mesmo

Inicialização

- Criar um nó cabeça
- O início da lista aponta para o nó cabeça
- O nó cabeça aponta para ele mesmo

```
No* iniciar_lista_cab(Lista *p_l){
    No *cab = malloc(sizeof(No));
    p_l->inicio = cab;
   cab->prox = cab;
   return cab;
}
```

Comprimento da lista

- Percorre a lista para contar a quantidade de elementos
  - Ignora-se o nó cabeça

Comprimento da lista

- Percorre a lista para contar a quantidade de elementos
  - Ignora-se o nó cabeça

```
int qtd_elementos(Lista *p_l){
    int qtd = 0;
    No* cab = p_l->inicio;
    No* aux = p_l->inicio->prox;
    while (aux!=cab){
        aux=aux->prox;
        qtd++;
    }
    return qtd;
}
```

Impressão

- Percorre cada elemento e lê o valor da chave
- Não considerar o nó cabeça (elemento não válido)

Impressão

- Percorre cada elemento e lê o valor da chave
- Não considerar o nó cabeça (elemento não válido)

```
void imprimir_lista_circular_cab(Lista *p_l) {
        //ignora o nó cabeça
        No *aux = p_l->inicio->prox;
        do {
                printf("%d ", aux->chave);
                aux = aux->prox;
        } while (aux != p_l->inicio);
}
```

Impressão

- Percorre cada elemento e lê o valor da chave
- Não considerar o nó cabeça (elemento não válido)

```
void imprimir_lista_circular_cab(Lista *p_l) {
        //ignora o nó cabeça
        No *aux = p_l->inicio->prox;
        do {
                printf("%d ", aux->chave);
                aux = aux->prox;
        } while (aux != p_l->inicio);
}
```

E se a lista estiver vazia?

Busca

- Nó auxiliar aponta para a cabeça
- Percorre lista procurando pelo valor buscado
- Se voltar ao nó cabeça ao final, elemento não foi encontrado, senão retorne endereço encontrado

Busca

- Nó auxiliar aponta para a cabeça
- Percorre lista procurando pelo valor buscado
- Se voltar ao nó cabeça ao final, elemento não foi encontrado, senão retorne endereço encontrado

```
No* busca_lista_circular_cab(Lista *p_l, int valor_ch) {
        No *aux = p_l->inicio;
        do {aux = aux->prox;
        } while (aux != p_l->inicio && aux->chave!=valor_ch);
        if (aux==p_l->inicio) return NULL;
        else return aux;
}
```

Busca com sentinela

- Adiciona o valor buscado ao nó cabeça
  - Auxilia na operação de busca
- Retorna o endereço do elemento buscado ou NULL

Busca com sentinela

- Adiciona o valor buscado ao nó cabeça
  - Auxilia na operação de busca
- Retorna o endereço do elemento buscado ou NULL

```
No* busca_lista_circular_cab_senti(Lista *p_l, int valor_ch) {
        p_l->inicio->chave=valor_ch;  //sentinela no Nó cabeça
        No *aux = p_l->inicio->prox;
        while(aux->chave!=valor_ch)
           aux = aux->prox;
        }
        if (aux!=p_l->inicio) return aux;
        return NULL;
}
```

Inserção

- Inserção ordenada pelo valor da chave
  - Não permite a inserção de elementos repetidos
- Identificar entre quais elementos o novo nó será inserido (alocar memória)
- É necessário conhecer o nó antecessor
  - Obter ponteiro para o mesmo

Procedimento auxiliar de busca para a inserção e remoção


```
No* busca_aux_lista_circ(Lista *p_l, int valor, No** ant) {
        No *corrente = p_l->inicio->prox; //pula o cabeça




}
```

Procedimento auxiliar de busca para a inserção e remoção


```
No* busca_aux_lista_circ(Lista *p_l, int valor, No** ant) {
        No *corrente = p_l->inicio->prox; //pula o cabeça
        p_l->inicio->chave=valor; //sentinela pelo cabeça




}
```

Procedimento auxiliar de busca para a inserção e remoção


```
No* busca_aux_lista_circ(Lista *p_l, int valor, No** ant) {
        No *corrente = p_l->inicio->prox; //pula o cabeça
        p_l->inicio->chave=valor; //sentinela pelo cabeça
        while(corrente->chave<valor) {
          (*ant)=corrente;
          corrente = corrente->prox;
        }



}
```

Procedimento auxiliar de busca para a inserção e remoção


```
No* busca_aux_lista_circ(Lista *p_l, int valor, No** ant) {
        No *corrente = p_l->inicio->prox; //pula o cabeça
        p_l->inicio->chave=valor; //sentinela pelo cabeça
        while(corrente->chave<valor) {
          (*ant)=corrente;
          corrente = corrente->prox;
        }
        if (corrente!=p_l->inicio && corrente->chave==chave)
           return corrente;
        return NULL;
}
```

Procedimento auxiliar de busca para a inserção e remoção


```
No* busca_aux_lista_circ(Lista *p_l, int valor, No** ant) {
        No *corrente = p_l->inicio->prox; //pula o cabeça
        p_l->inicio->chave=valor; //sentinela pelo cabeça
        while(corrente->chave<valor) {
          (*ant)=corrente;
          corrente = corrente->prox;
        }
        if (corrente!=p_l->inicio && corrente->chave==chave)
           return corrente;
        return NULL;
}
```

Este código funciona para todos os casos?

Procedimento auxiliar de busca para a inserção e remoção


```
No* busca_aux_lista_circ(Lista *p_l, int valor, No** ant) {
        No *corrente = p_l->inicio->prox; //pula o cabeça
        p_l->inicio->chave=valor; //sentinela pelo cabeça
        while(corrente->chave<valor) {
          (*ant)=corrente;
          corrente = corrente->prox;
        }
        if (corrente!=p_l->inicio && corrente->chave==chave)
           return corrente;
        return NULL;
}
```

Este código funciona para todos os casos?

Elemento buscado for menor que o primeiro. (*ant)?

Inserção


```
bool insere_lista_circular_cab(Lista *p_l, int valor) {
        No* ant, novo;
        novo = busca_aux_lista_circ(p_l, valor, &ant)
        if (novo!=NULL)
            return false; //elemento já existe
        novo = malloc(sizeof(No); //aloca memória
        novo->chave=chave;
        //inserção no meio
        novo->prox=ant->prox;
        ant->prox=novo;
        return true;
}
```

Inserção


```
bool insere_lista_circular_cab(Lista *p_l, int valor) {
        No* ant, novo;
        novo = busca_aux_lista_circ(p_l, valor, &ant)
        if (novo!=NULL)
            return false; //elemento já existe
        novo = malloc(sizeof(No); //aloca memória
        novo->chave=chave;
        //inserção no meio
        novo->prox=ant->prox;
        ant->prox=novo;
        return true;
}
```

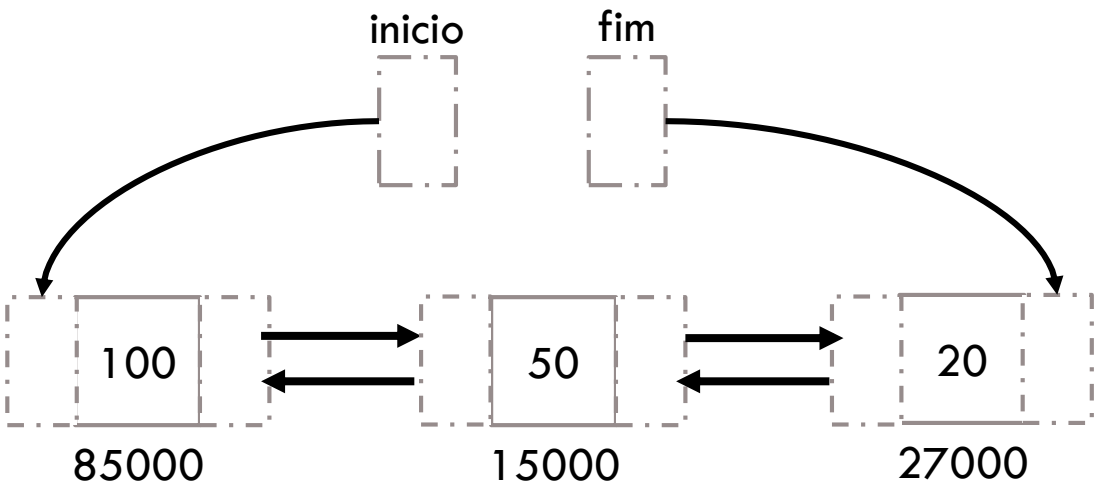
Se ant == NULL?

Remoção


```
bool remove_lista_circular_cab(Lista *p_l, int valor) {
        No* ant, excl;
        excl = busca_aux_lista_circ(p_l, valor, &ant)
        if (excl==NULL)
                return false; //elemento não existe
        ant->prox=excl->prox;
        free(excl);
        return true;
}
```

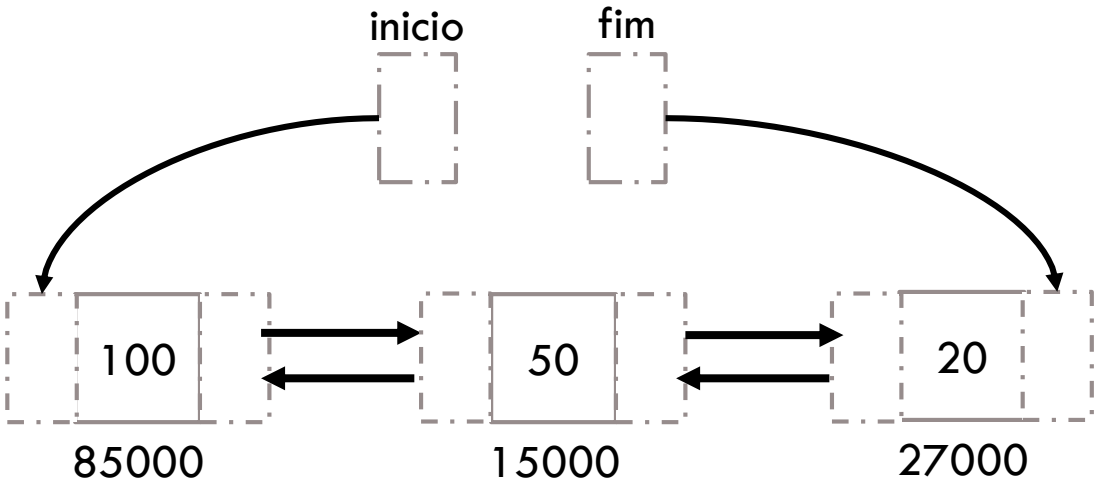
Lista duplamente ligada

- Percorre em duas direções



Lista duplamente ligada

- Percorre em duas direções



- Lista vazia

Estrutura da lista duplamente ligada

Estrutura do Nó

```
typedef struct NoDupl{
      int chave;
     // outros campos...
     struct No *esq;
     struct No *dir;
} No;
```

Lista duplamente ligada

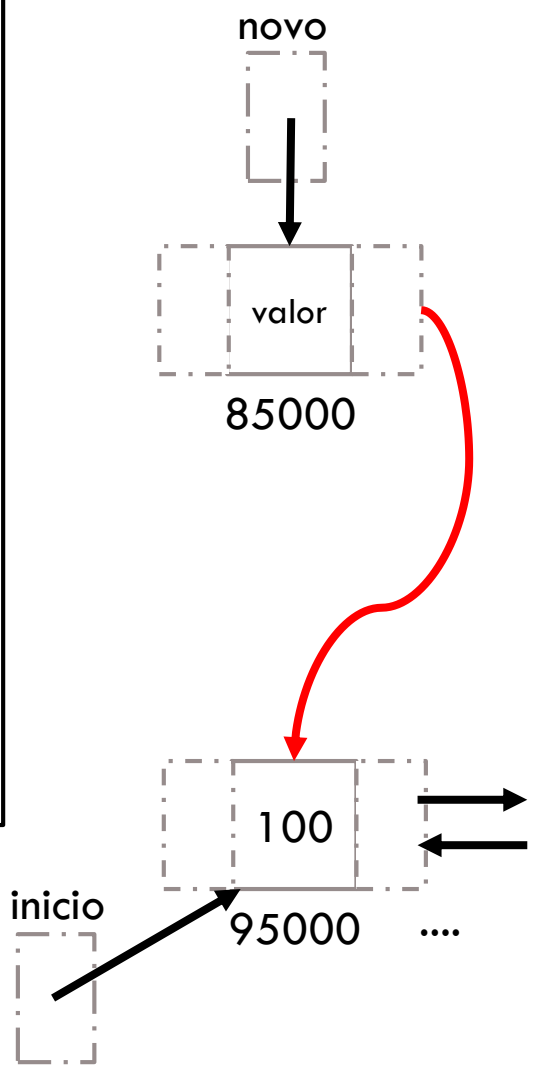
```
typedef struct {
      No *inicio;
      No *fim;
} ListaDupl;
```

Inserção em lista duplamente ligada


```
bool insere_lista_dupl(ListaDupl *pl, int valor) {
        NoDupl* novo;
        novo = malloc(sizeof(NoDupl); //aloca memória
        if (novo==NULL)
                return false;
        novo->chave=valor;




}
```

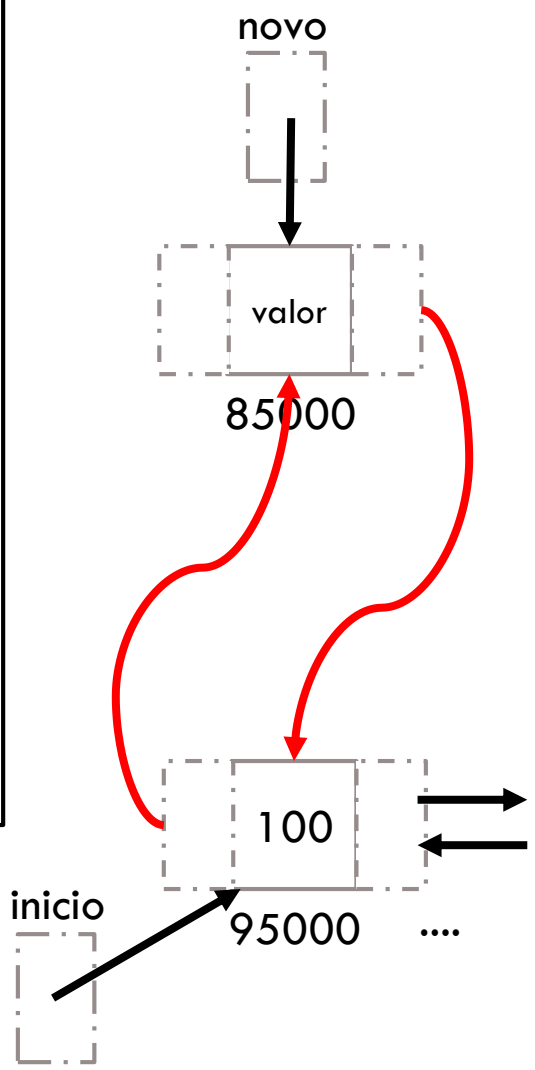
Inserção em lista duplamente ligada


```
bool insere_lista_dupl(ListaDupl *pl, int valor) {
        NoDupl* novo;
        novo = malloc(sizeof(NoDupl);
        if (novo==NULL)
                return false;
        novo->chave=valor;
        novo->dir=pl->inicio;



}
```



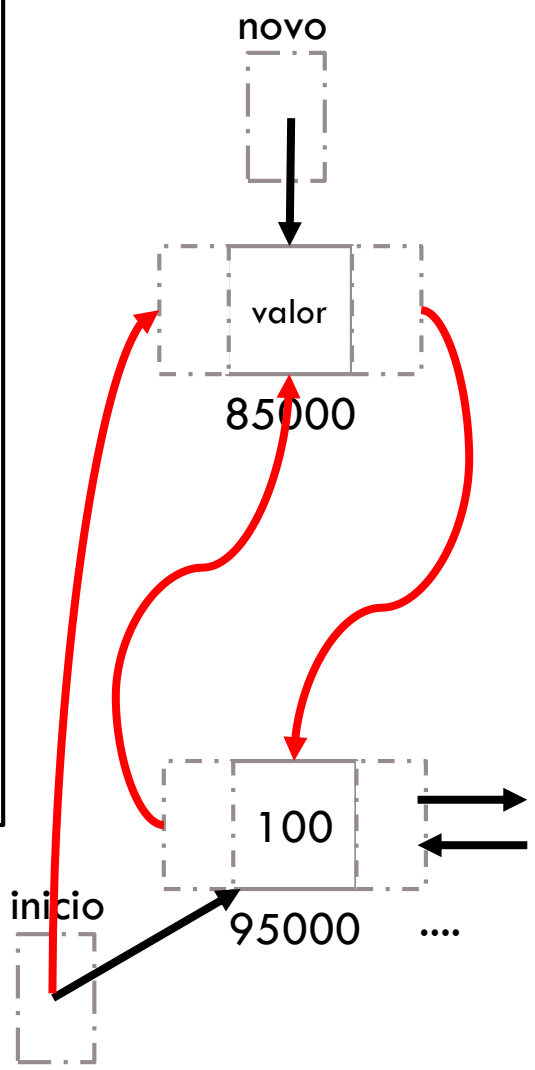
Inserção em lista duplamente ligada


```
bool insere_lista_dupl(ListaDupl *pl, int valor) {
        NoDupl* novo;
        novo = malloc(sizeof(NoDupl);
        if (novo==NULL)
                return false;
        novo->chave=valor;
        novo->dir=pl->inicio;
        pl->inicio->esq=novo;


}
```



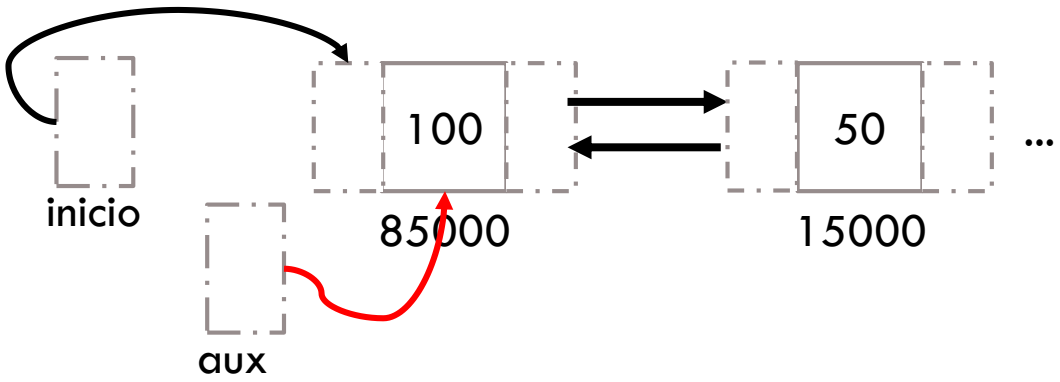
Inserção em lista duplamente ligada


```
bool insere_lista_dupl(ListaDupl *pl, int valor) {
        NoDupl* novo;
        novo = malloc(sizeof(NoDupl);
        if (novo==NULL)
                return false;
        novo->chave=valor;
        novo->dir=pl->inicio;
        pl->inicio->esq=novo;
        pl->inicio=novo;
        return true;
}
```



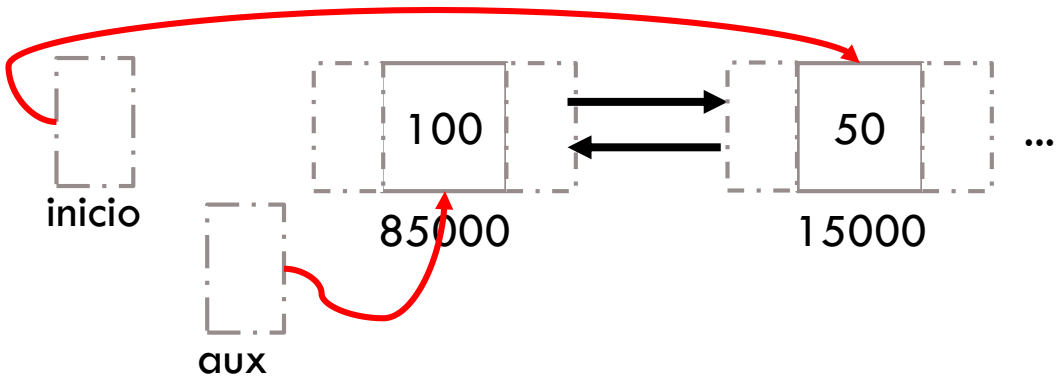
Remoção em lista duplamente ligada


```
bool remove_lista_dupl(ListaDupl *pl) {
        NoDupl* aux;
        aux=pl->inicio;




}
```



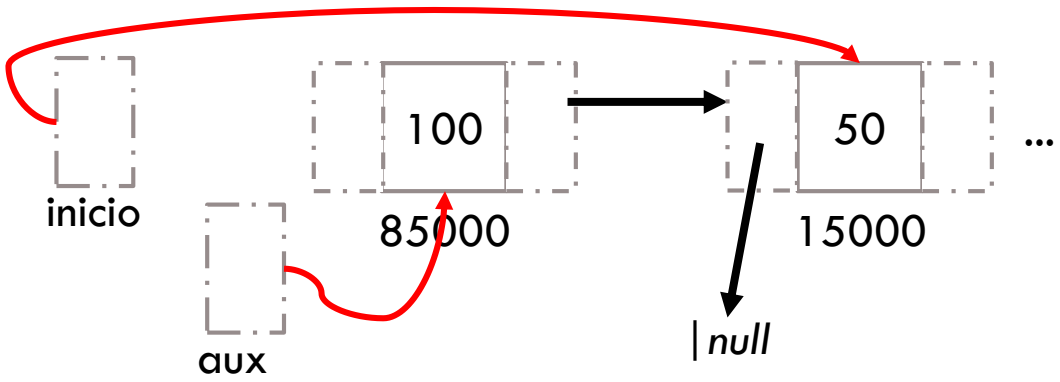
Remoção em lista duplamente ligada


```
bool remove_lista_dupl(ListaDupl *pl) {
        NoDupl* aux;
        aux=pl->inicio;
        pl->inicio=pl->inicio->dir;



}
```



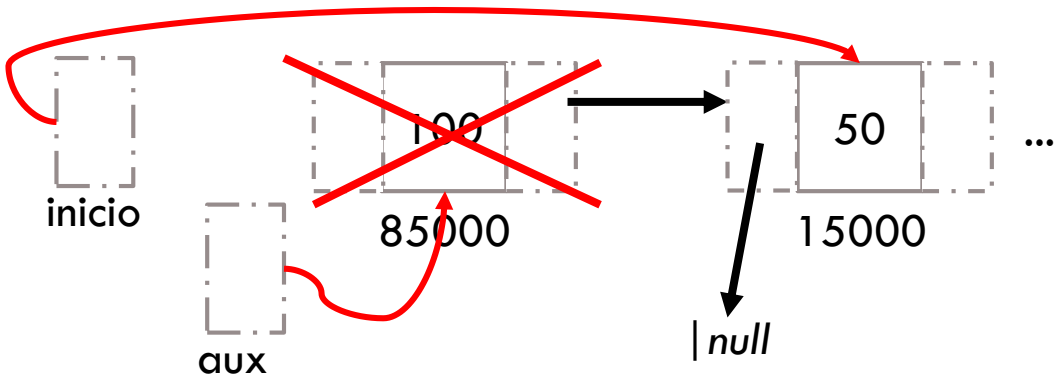
Remoção em lista duplamente ligada


```
bool remove_lista_dupl(ListaDupl *pl) {
        NoDupl* aux;
        aux=pl->inicio;
        pl->inicio=pl->inicio->dir;
        pl->inicio->esq=NULL;


}
```

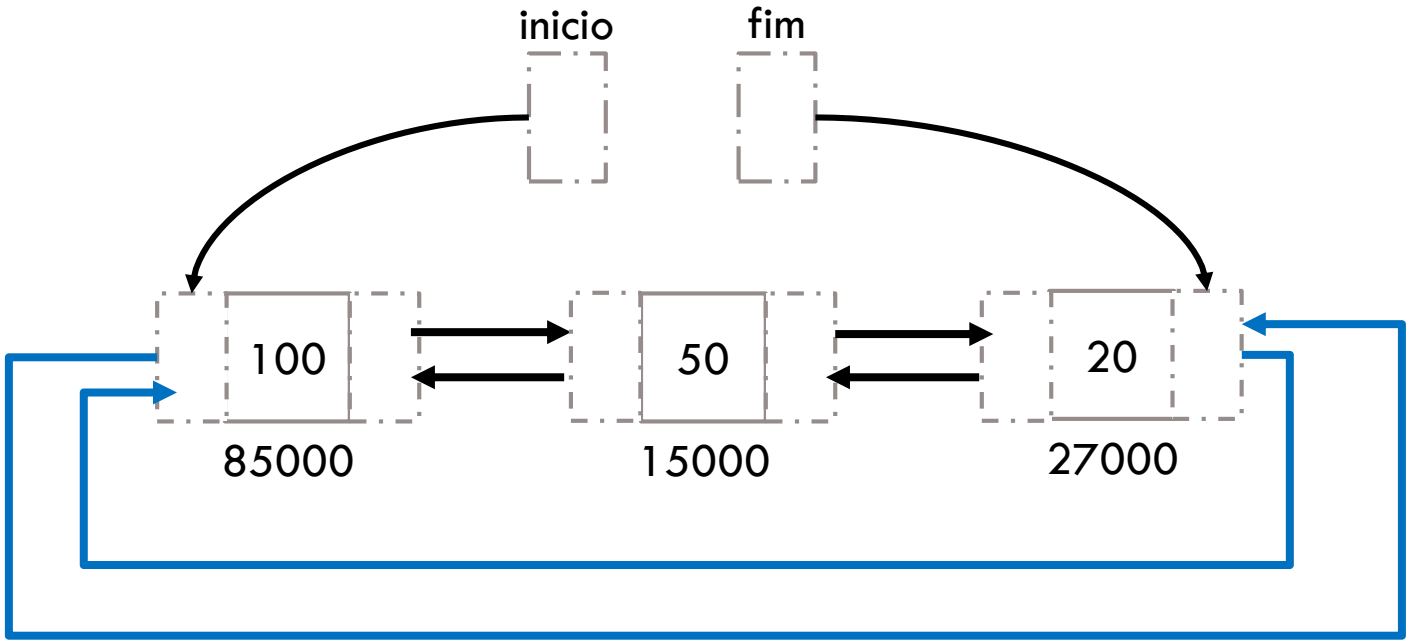


Remoção em lista duplamente ligada

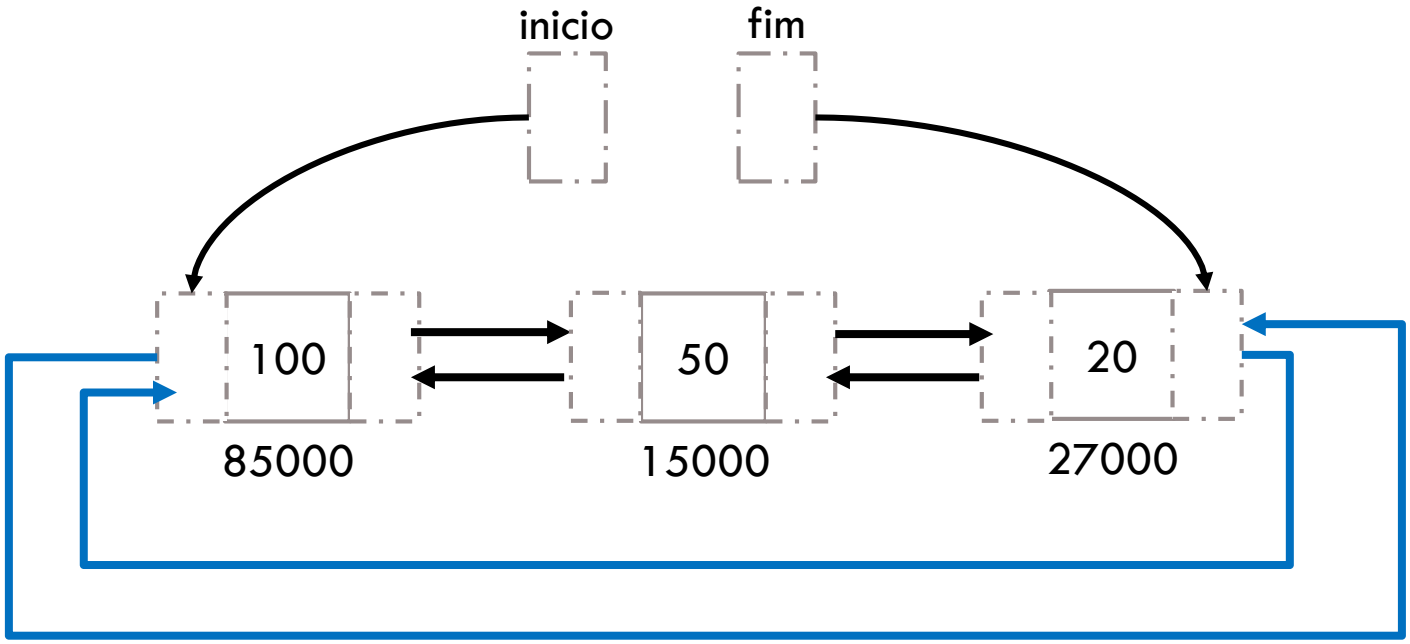

```
bool remove_lista_dupl(ListaDupl *pl) {
        NoDupl* aux;
        aux=pl->inicio;
        pl->inicio=pl->inicio->dir;
        pl->inicio->esq=NULL;
        free(aux);
        return true;
}
```



Duplamente ligada circular

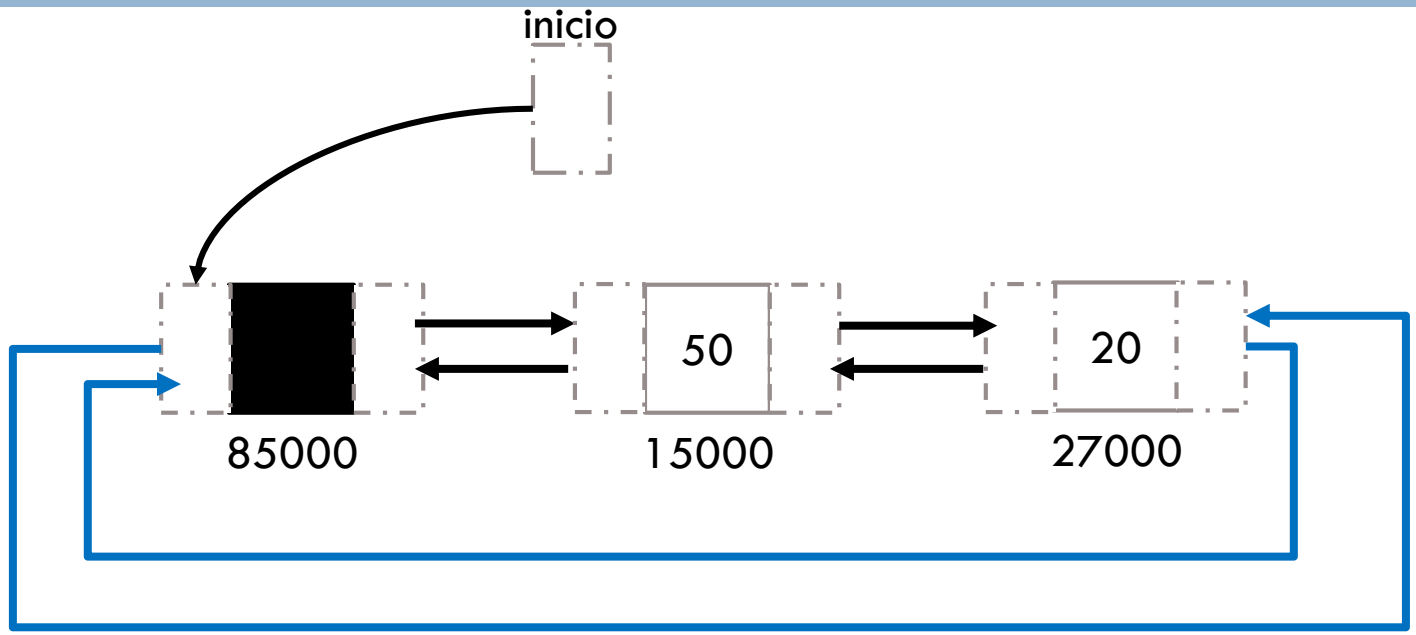


Duplamente ligada circular

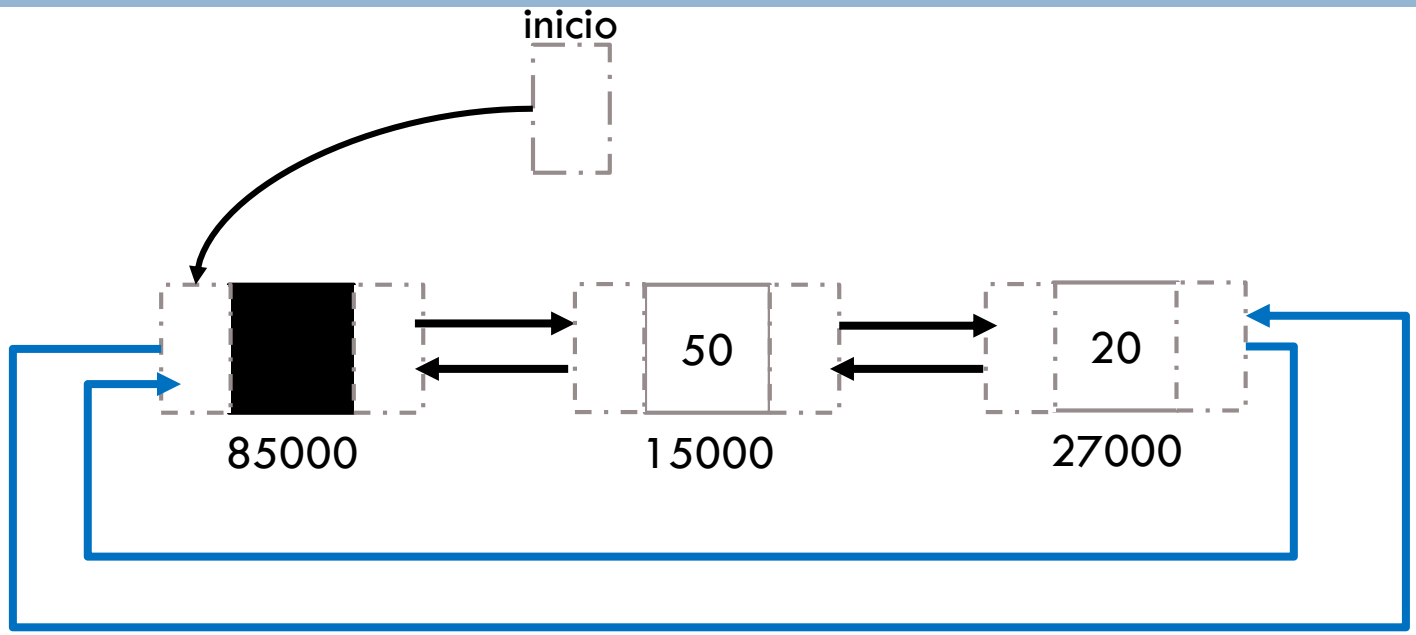



```
ListaDupl *lp;
lp->inicio->esq=lp->fim;
lp->fim->dir=lp->inicio;
```

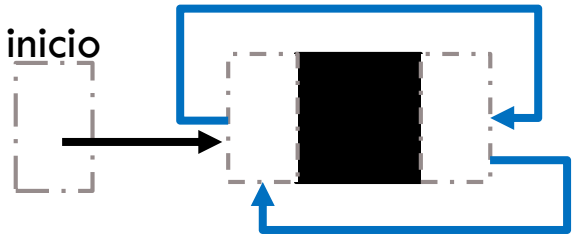
Duplamente ligada com cabeça



Duplamente ligada com cabeça



- Lista vazia



Síntese

- Diferentes variações para o projeto de estruturas ligadas
- Permitem facilitar aplicações de listas em contextos específicos
  - Necessidade de acessar mais facilmente os elementos
- Requerem implementações específicas para considerar nó cabeça, estrutura circular e dupla ligação

Noções de eficiência de algoritmos

Acessando um elemento na lista

- Em um arranjo

```
int val = vet[pos];
```

  - Número de operações é constante
    - Independente do tamanho do arranjo

Acessando um elemento na lista

- Em um arranjo

```
int val = vet[pos];
```

  - Número de operações é constante
    - Independente do tamanho do arranjo

- Em uma estrutura ligada

```
int i=0;
while(i<pos){
   p = p->prox;    i++;
}
int val = p->chave;
```

  - Número de operações é proporcional ao tamanho de *pos*

Contando instruções (análise simplificada)


```
int val = a + b;
```

Número de instruções: 1

Contando instruções (análise simplificada)


```
int val = a + b;
```

Número de instruções: 1

```
for (i=0; i<n; i++)
    val = a+b;
```

Número de instruções: n

Contando instruções (análise simplificada)


```
int val = a + b;
```

Número de instruções: 1

```
for (i=0; i<n; i++)
    val = a+b;
```

Número de instruções: n

```
for (i=0; i<n; i++)
   for (k=0; k<n; k++)
        val = a+b;
```

Número de instruções: n^2

Análise de eficiência de algoritmos

- Identificar uma função que expressa a complexidade do algoritmo
  - Com base em um valor de entrada
- Permite verificar a velocidade de execução do algoritmo de maneira teórica
- Análise do comportamento do algoritmo
  - Comparação da eficiência de algoritmos

Análise de eficiência de algoritmos

- Notação O
  - Uma função f(n) é da ordem de g(n), ou f(n) = O(g(n)), se:
    - Existe uma constante c
    - Existe uma constante n_0

    tal que, para todo $n >= n_0$

Análise de eficiência de algoritmos

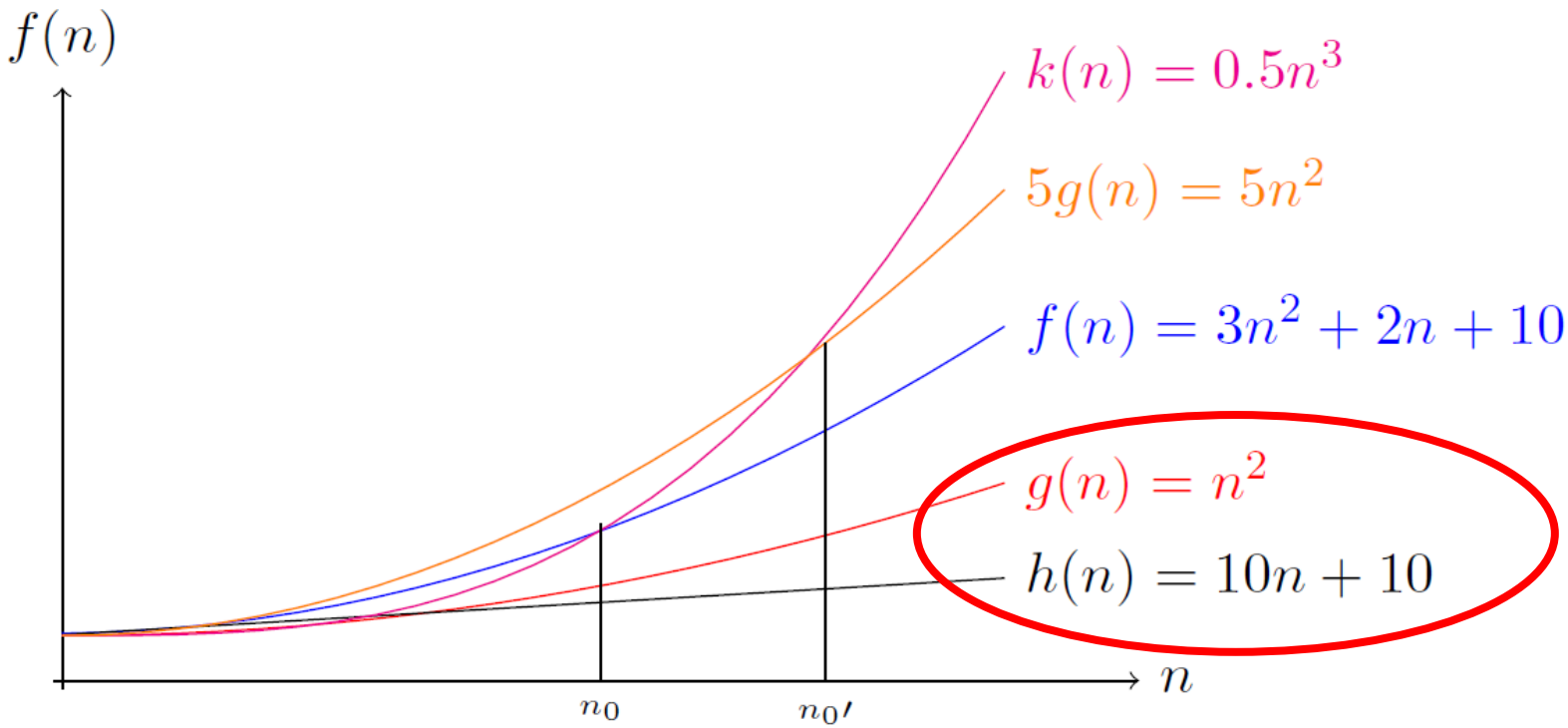
- Notação O
  - Uma função f(n) é da ordem de g(n), ou f(n) = O(g(n)), se:
    - Existe uma constante c
    - Existe uma constante n_0

    tal que, para todo $n >= n_0$

$$f(n) <= c * g(n)$$

  - O crescimento de g(n) domina o crescimento de f(n) acima de n_0

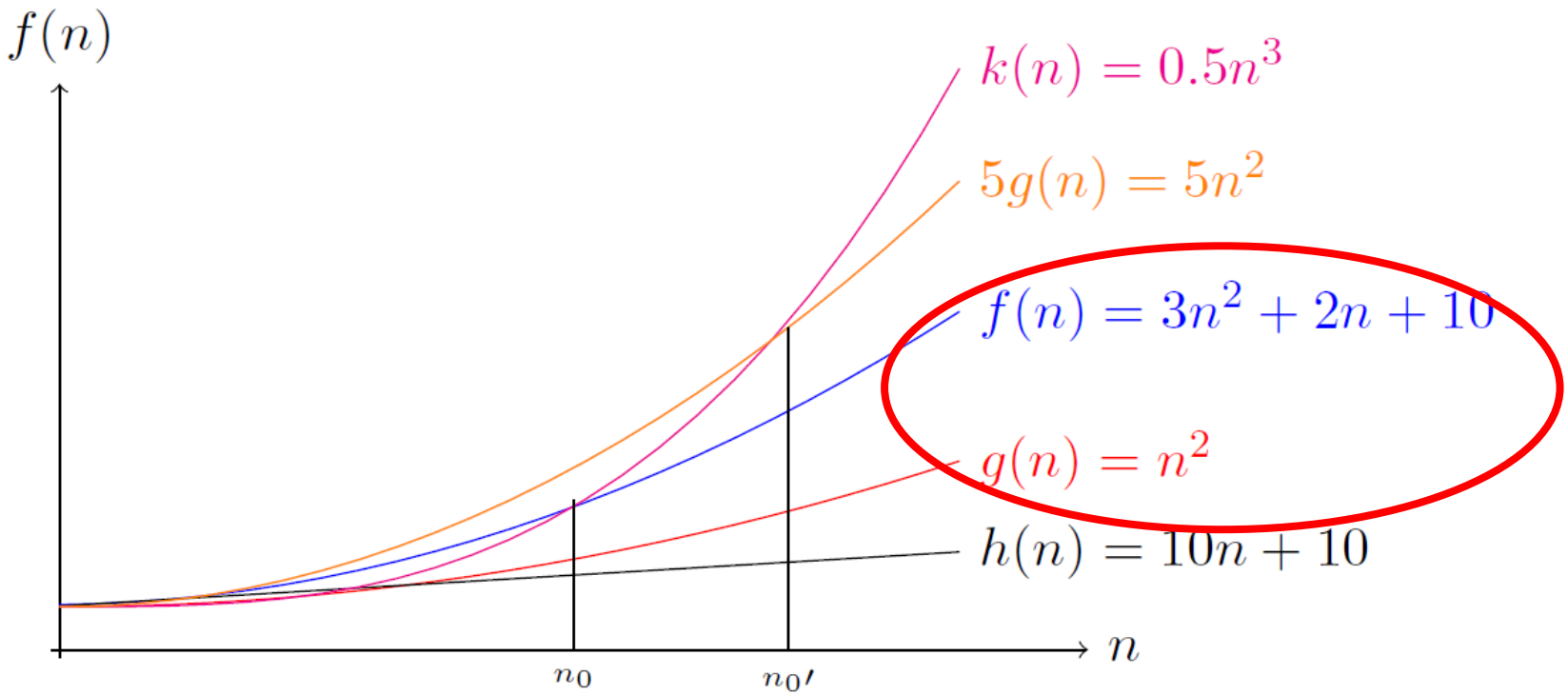
Análise de eficiência de algoritmos *



h(n) = O(g(n))

*Exemplo retirado do material de Tomasz Kowaltowski (Estruturas de Dados e Técnicas de Programação)

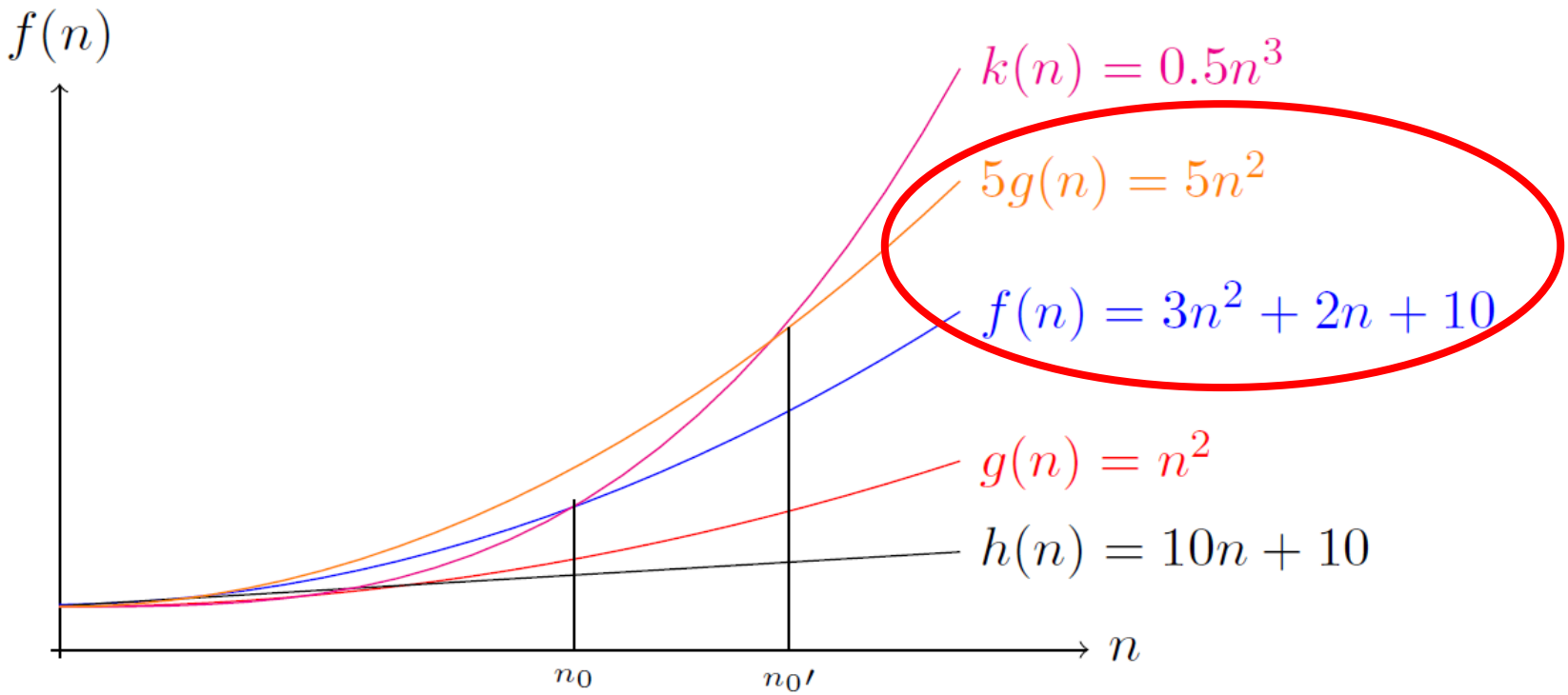
Análise de eficiência de algoritmos *



g(n) = O(f(n))

*Exemplo retirado do material de Tomasz Kowaltowski (Estruturas de Dados e Técnicas de Programação)

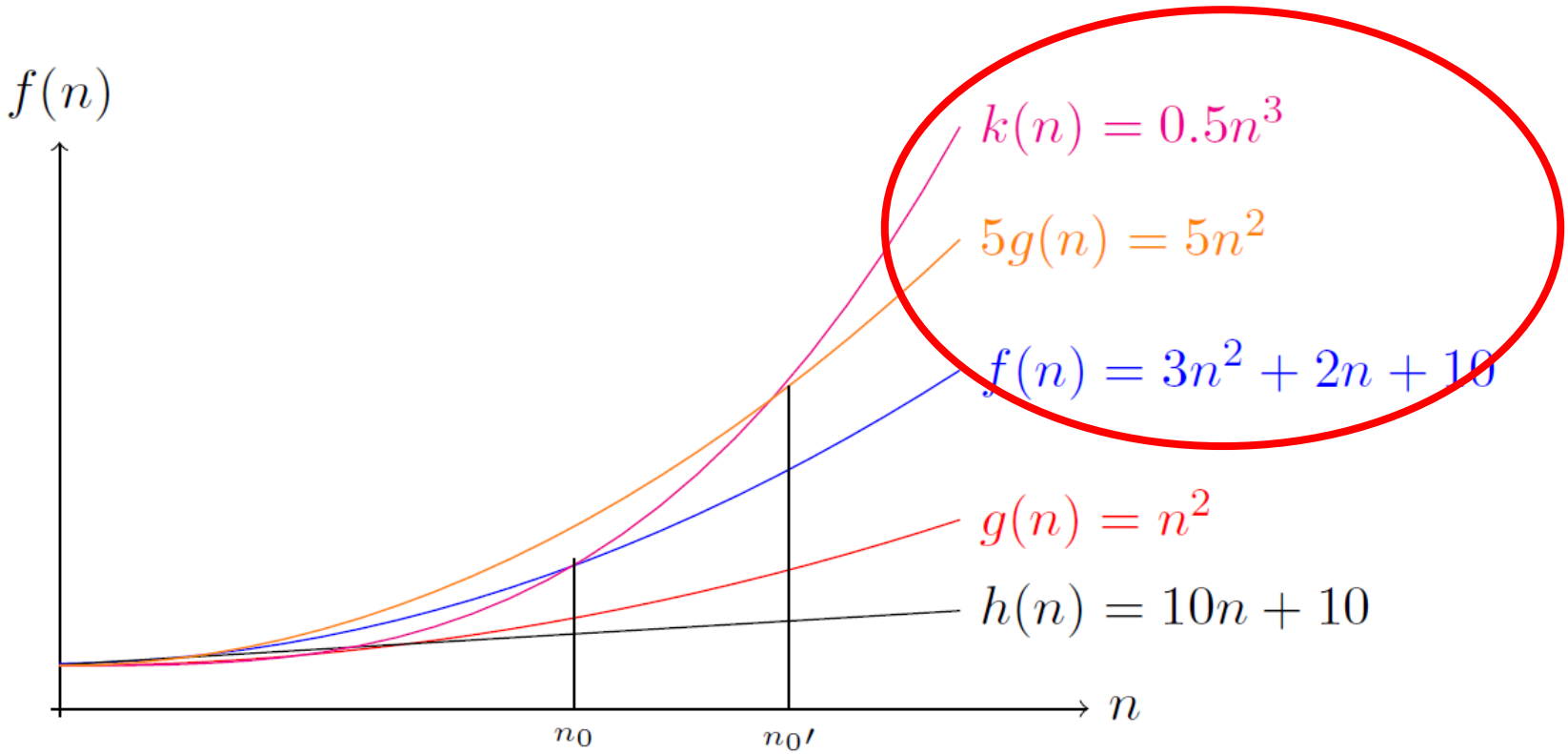
Análise de eficiência de algoritmos *



f(n) = O(g(n))

*Exemplo retirado do material de Tomasz Kowaltowski (Estruturas de Dados e Técnicas de Programação)

Análise de eficiência de algoritmos *



f(n) = O(k(n))

*Exemplo retirado do material de Tomasz Kowaltowski (Estruturas de Dados e Técnicas de Programação)

Exemplos

- 1 = O(1)
- 1.000.000 = O(1)

c = O(1)
para qualquer constante c

Exemplos

- $1 = O(1)$
- $1.000.000 = O(1)$
- $5n + 2 = O(n)$
- $5n^2 + 5n + 2 = O(n^2)$
- $1.000.000n = O(n^2)$

$c = O(1)$ para qualquer constante c

Exemplos

- $1 = O(1)$
- $1.000.000 = O(1)$

$c = O(1)$ para qualquer constante c

- $5n + 2 = O(n)$
- $5n^2 + 5n + 2 = O(n^2)$
- $1.000.000n = O(n^2)$
- $10n^2 = O(n^3)$

Exemplos

- $1 = O(1)$
- $1.000.000 = O(1)$

$c = O(1)$ para qualquer constante c

- $5n + 2 = O(n)$
- $5n^2 + 5n + 2 = O(n^2)$
- $1.000.000n = O(n^2)$
- $10n^2 = O(n^3)$
- $\log_2 n = O(\log_{10} n)$
- $\log_{10} n = O(\log_2 n)$

$\text{Log}_a\ n = O(\log_b n)$ para $a, b > 0$

Exemplos

- $1 = O(1)$
- $1.000.000 = O(1)$

$c = O(1)$ para qualquer constante c

- $5n + 2 = O(n)$
- $5n^2 + 5n + 2 = O(n^2)$
- $1.000.000n = O(n^2)$
- $10n^2 = O(n^3)$
- $\log_2 n = O(\log_{10} n)$
- $\log_{10} n = O(\log_2 n)$

$\text{Log}_a\ n = O(\log_b n)$ para $a, b > 0$

- $n^{1000} = O(2^n)$

Terminologia (notação assintótica)

- O(1): constante
  - Tempo não depende do valor de *n*
- O(n): crescimento linear
  - Quando *n* dobra, o tempo dobra

Terminologia (notação assintótica)

- O(1): constante
  - Tempo não depende do valor de *n*
- O(n): crescimento linear
  - Quando *n* dobra, o tempo dobra
- O(log n): crescimento logarítmico
  - Quando *n* dobra, o tempo aumenta em 1
- O(n log n): crescimento n-log-n
  - Quando *n* dobra, o tempo um pouco mais que dobra

Terminologia (notação assintótica)

- $O(n^2)$: crescimento quadrático
  - quando *n* dobra, o tempo quadriplica

Terminologia (notação assintótica)

- $O(n^2)$: crescimento quadrático
  - quando *n* dobra, o tempo quadriplica
- $O(n^3)$: crescimento cúbico
  - quando *n* dobra, o tempo octuplica
- $O(2^n)$: crescimento exponencial

Comparação de execuções*

- Busca sequencial – O(n) vs. Busca binária – O(log n)

- Duas máquinas (suposição)
  - MR – muito rápida
  - ML – muito lenta (ao menos 100 vezes mais lenta)

- MR executa busca sequencial
- ML executa busca binária

*Exemplo retirado do material de Tomasz Kowaltowski (Estruturas de Dados e Técnicas de Programação)

Comparação de execuções*

n	$\log_2 n$	M_R (busca sequencial – $O(n)$)	M_L (busca binária – $O(\log n)$)
16	4	16	400
32	5	32	500
64	6	64	600
128	7	128	700
256	8	256	800
512	9	512	900
1024	10	1024	1000
2048	11	2048	1100
4096	12	4096	1200
...	...	...	...
2^{20}	20	1.048.576	2000
2^{21}	21	2.097.152	2100
...	...	...	...
2^{30}	30	1.073.741.824	3000

*Exemplo retirado do material de Tomasz Kowaltowski (Estruturas de Dados e Técnicas de Programação)

Listas com arranjos vs. listas ligadas

- Acesso à uma posição
  - Direito pelo índice do arranjo : O(1)
  - Lista ligada exige percorrer os elementos : O(n)

Listas com arranjos vs. listas ligadas

- Acesso à uma posição
  - Direito pelo índice do arranjo : O(1)
  - Lista ligada exige percorrer os elementos : O(n)

- Inserção em uma posição
  - No arranjo é necessário deslocar elementos : O(n)
  - Na lista ligada, se houver ponteiro para o nó predecessor, apenas os ponteiros precisam ser atualizados : O(1)

Listas com arranjos vs. listas ligadas

- Acesso à uma posição
  - Direito pelo índice do arranjo : O(1)
  - Lista ligada exige percorrer os elementos : O(n)
- Inserção em uma posição
  - No arranjo é necessário deslocar elementos : O(n)
  - Na lista ligada, se houver ponteiro para o nó predecessor, apenas os ponteiros precisam ser atualizados : O(1)
- Remoção em uma posição
  - No arranjo é necessário deslocar elementos : O(n)
  - Atualização de ponteiros, se predecessor é conhecido : O(1)

Listas com arranjos vs. listas ligadas

- Uso de espaço de memória
  - Com arranjos
    - Pode causar desperdício de espaço
      - Difícil conhecer o limitante para um tamanho máximo
  - Com ponteiros
    - Maior flexibilidade, mas cada elemento exige mais espaço para armazenar os ponteiros