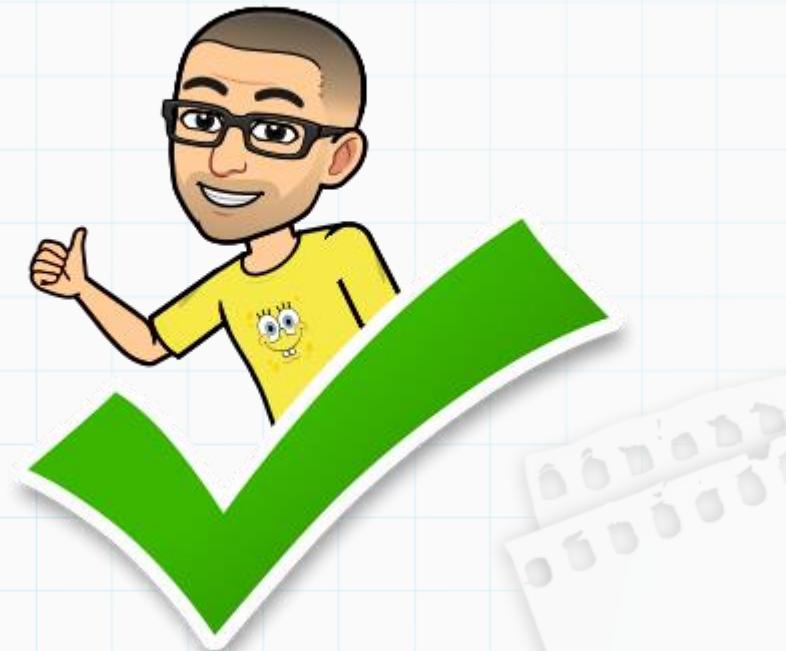
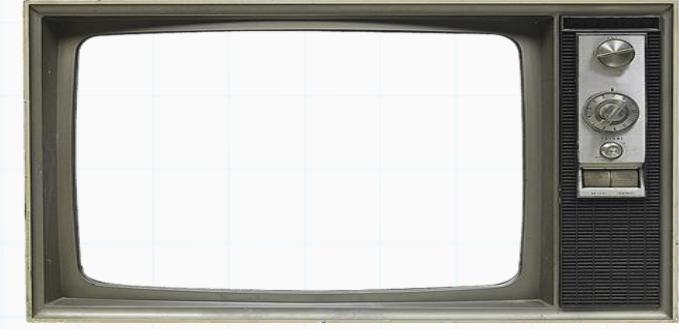


# Programação Estruturada

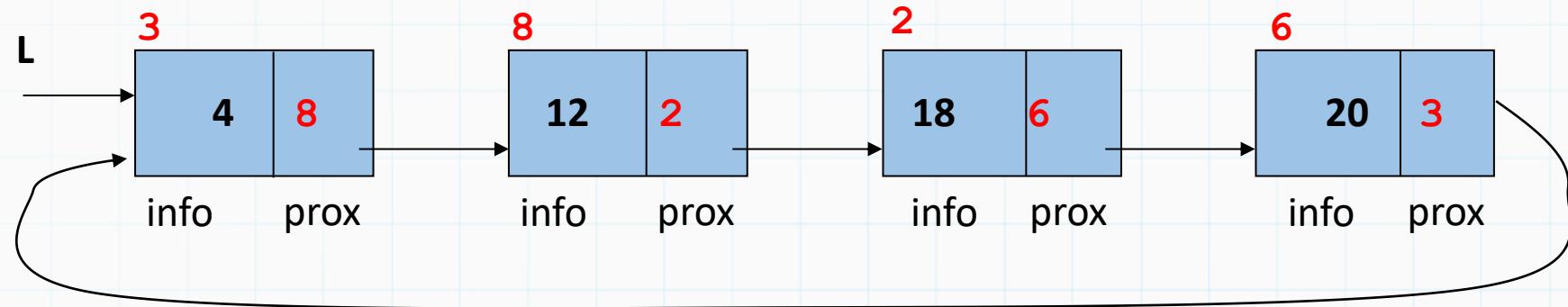
Professor : Yuri Frota

yuri@ic.uff.br



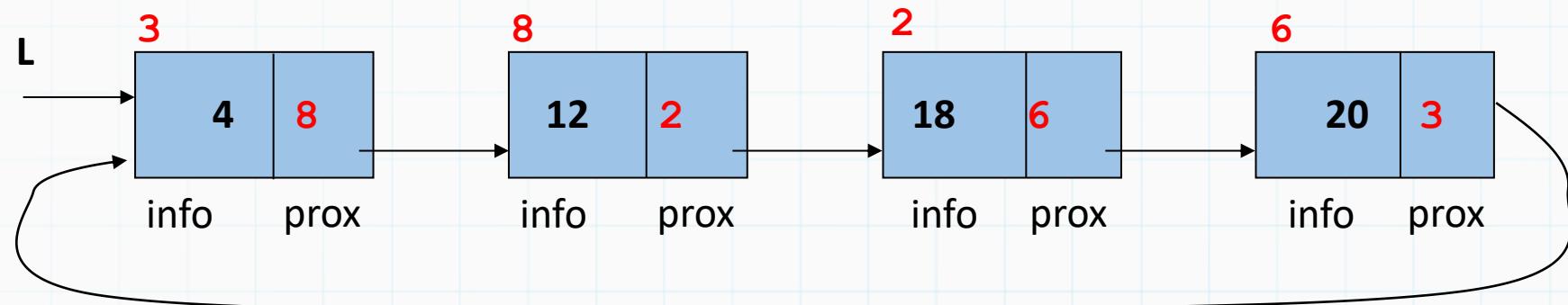
# Listas Circulares

É uma lista circular encadeada, o último nó da lista aponta para o primeiro (na verdade não existe nem último nem primeiro pois é circular) Dizemos que o ponteiro aponta para a cabeça da lista, e não para o inicio.



# Listas Circulares

É uma lista circular encadeada, o último nó da lista aponta para o primeiro (na verdade não existe nem último nem primeiro pois é circular) Dizemos que o ponteiro aponta para a cabeça da lista, e não para o inicio.

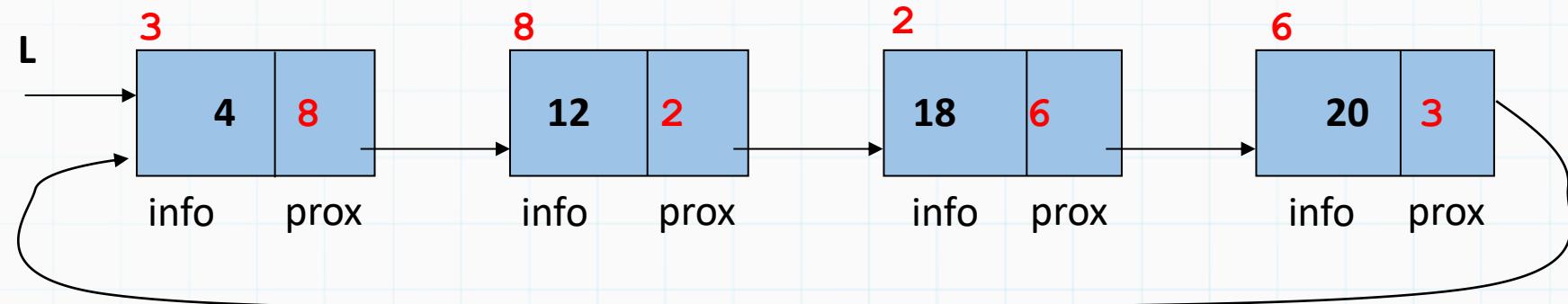


## Vantagens:

- São vantajosas para operações onde o fim e o inicio coincidem, e métodos que precisam ficar ciclando em grupos de elementos:
  - Ex: atribuição de tarefas a um processador, onde se uma tarefa não foi processada por não está pronta, ela tem que esperar a próxima rodada

# Listas Circulares

É uma lista circular encadeada, o último nó da lista aponta para o primeiro (na verdade não existe nem último nem primeiro pois é circular) Dizemos que o ponteiro aponta para a cabeça da lista, e não para o inicio.



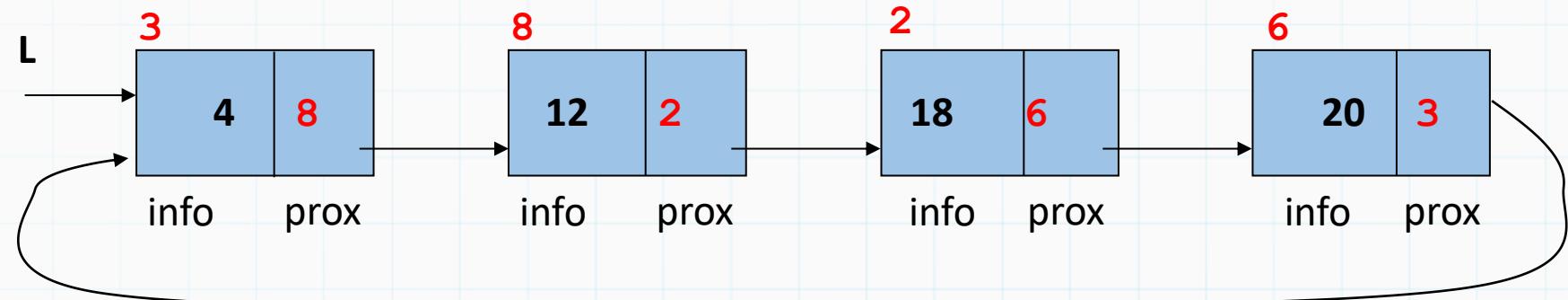
## Vantagens:

- São vantajosas para operações onde o fim e o inicio coincidem, e métodos que precisam ficar ciclando em grupos de elementos:
  - Ex: atribuição de tarefas a um processador, onde se uma tarefa não foi processada por não está pronta, ela tem que esperar a próxima rodada
- **Desvantagens:**
  - São mais complexas

# Listas Circulares

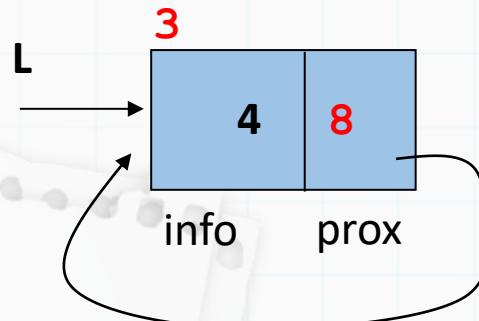


É uma lista circular encadeada, o último nó da lista aponta para o primeiro (na verdade não existe nem último nem primeiro pois é circular) Dizemos que o ponteiro aponta para a cabeça da lista, e não para o inicio.



Exemplos:

Lista com um elemento

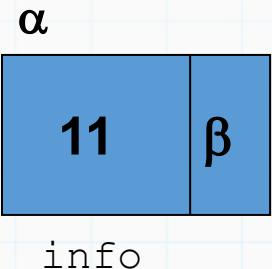


Lista vazia

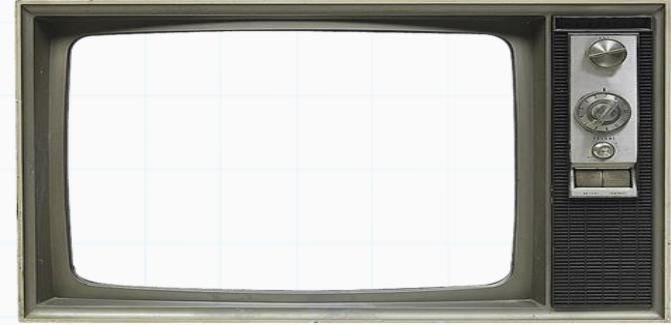


# Listas Circulares

A estrutura da lista não muda, apenas sua manipulação



```
struct NO {  
    int info;  
    struct NO *prox;  
}  
typedef struct NO lista;  
  
...  
lista *L;  
L = (lista*) malloc(sizeof(lista));  
L->prox = L;
```

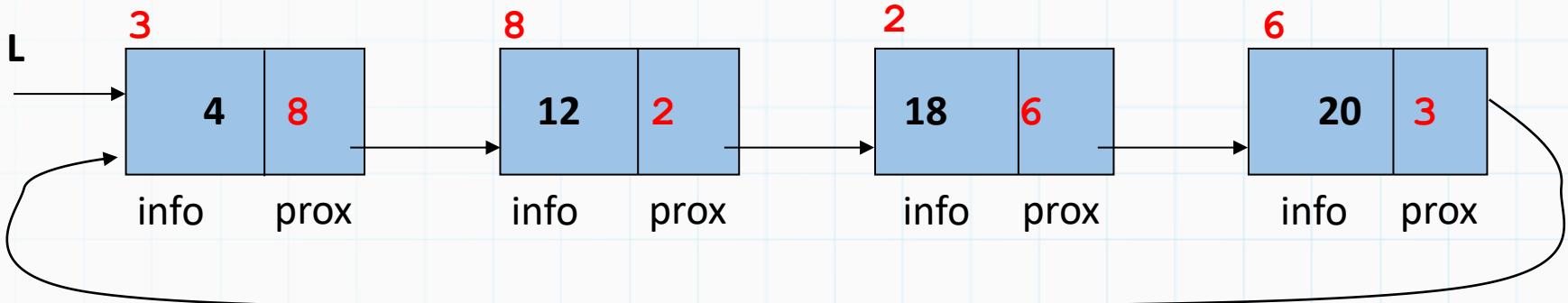


Tratamento especial para o  
primeiro elemento alocado: ele  
aponta para ele mesmo

# Listas Circulares

Percorrer: o que mudaria na função de imprimir a lista em uma lista circular ?

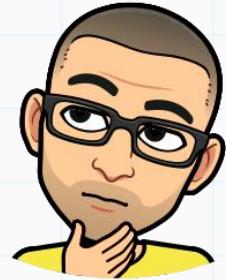
```
lista *no;
no = L;
while (no != NULL)
{
    printf("%d, ", no->info);
    no = no->prox;
}
```



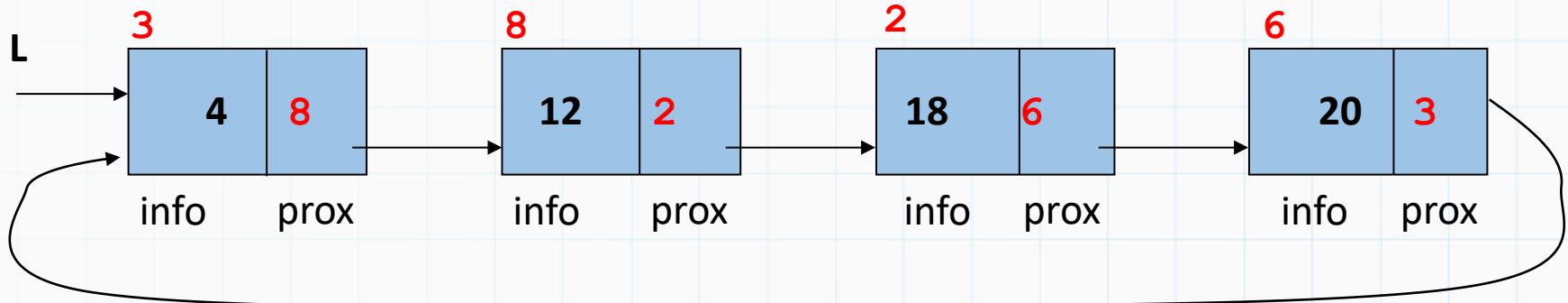
# Listas Circulares

Percorrer: o que mudaria na função de imprimir a lista em uma lista circular ?

```
lista *no;
no = L;
while (no != L)
{
    printf("%d, ", no->info);
    no = no->prox;
}
```



Da para fazer assim ?



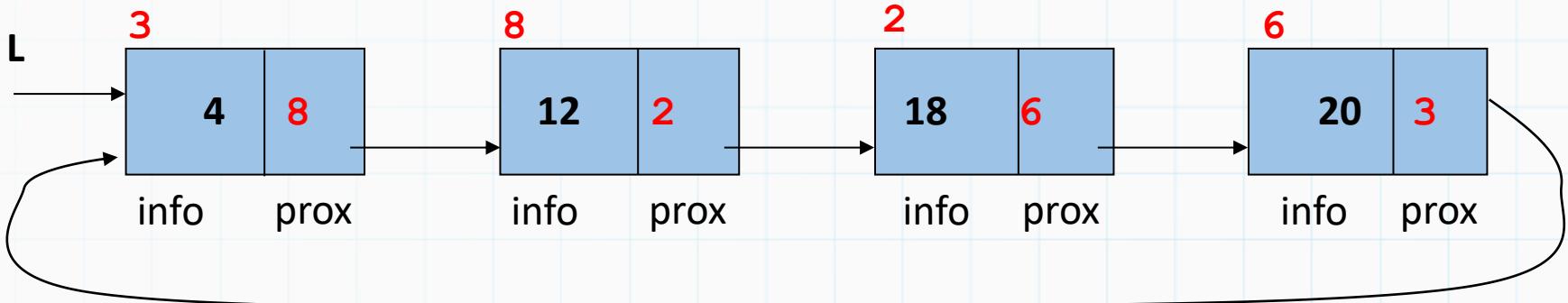
# Listas Circulares

Percorrer: o que mudaria na função de imprimir a lista em uma lista circular ?

```
lista *no;
no = L;
while (no != L)
{
    printf("%d, ", no->info);
    no = no->prox;
}
```



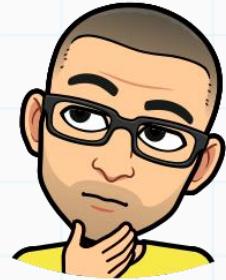
nem vai entrar



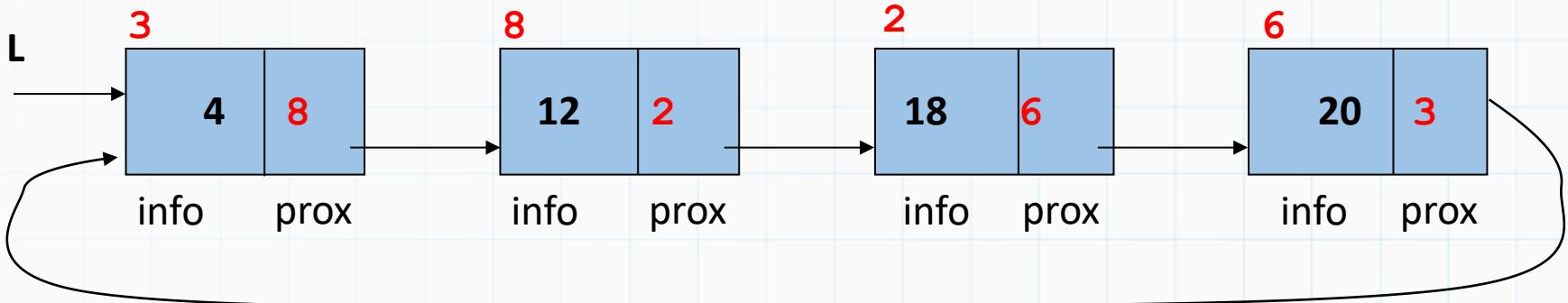
# Listas Circulares

Percorrer: o que mudaria na função de imprimir a lista em uma lista circular ?

```
lista *no;
no = L;
do
{
    printf("%d, ", no->info);
    no = no->prox;
} while (no != L);
```



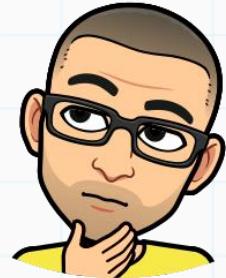
Da para fazer assim ?



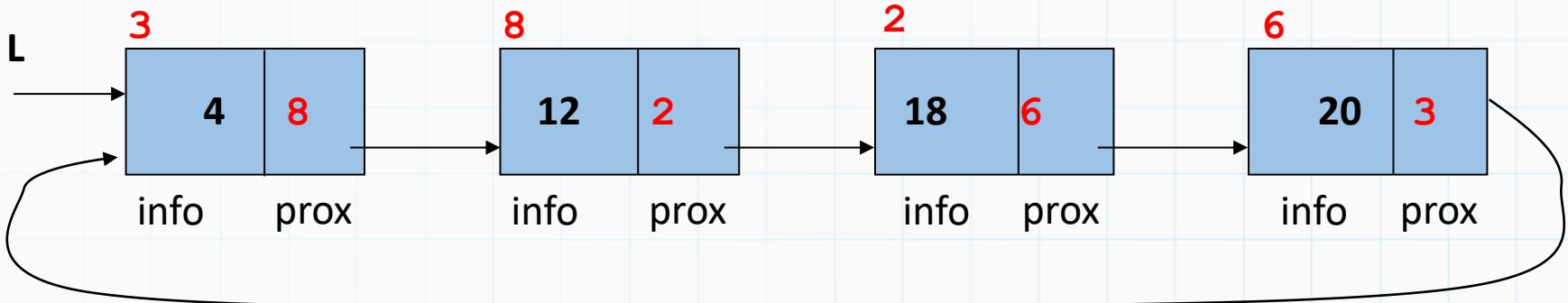
# Listas Circulares

Percorrer: o que mudaria na função de imprimir a lista em uma lista circular ?

```
lista *no;
no = L;
do
{
    printf ("%d, ", no->info);
    no = no->prox;
} while (no != L);
```



mas e se L vazia ?



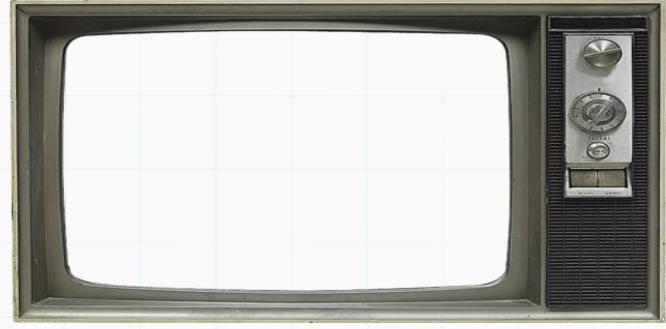
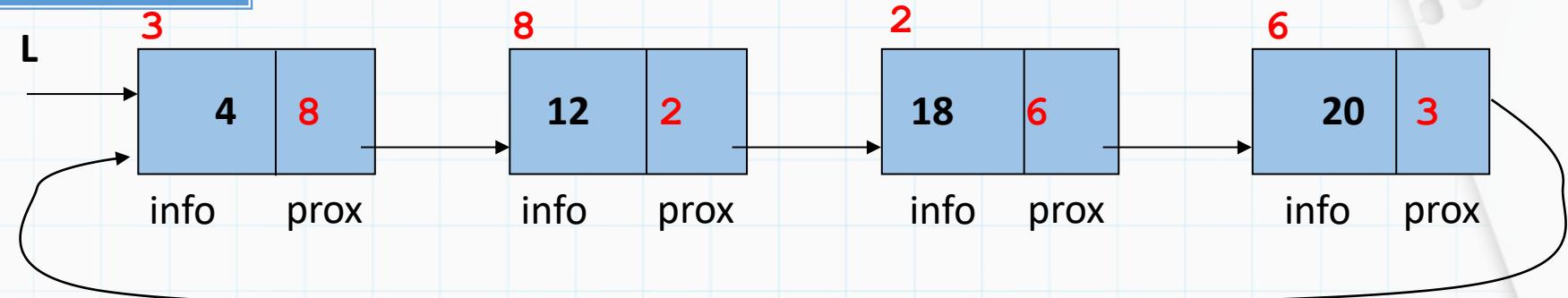
# Listas Circulares

Percorrer: o que mudaria na função de imprimir a lista em uma lista circular ?

```
lista *no;
no = L;

if (L == NULL)
{
    printf("L = vazio");
    return;
}

do
{
    printf("%d, ", no->info);
    no = no->prox;
} while (no != L);
```



# Listas Circulares

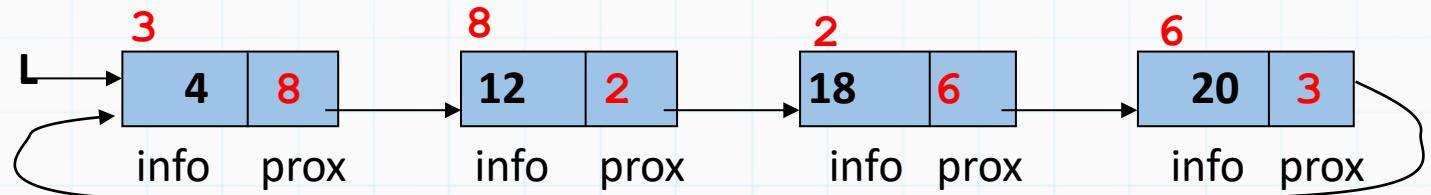
Outro exemplo de percorrer: Vamos percorrer para contar quantos elementos ?

```
int tamanho(lista* L)
{
    if (L == NULL)
        return 0;

    lista* no = L;
    int tam = 0;
    do
    {
        tam++;
        no = no->prox;

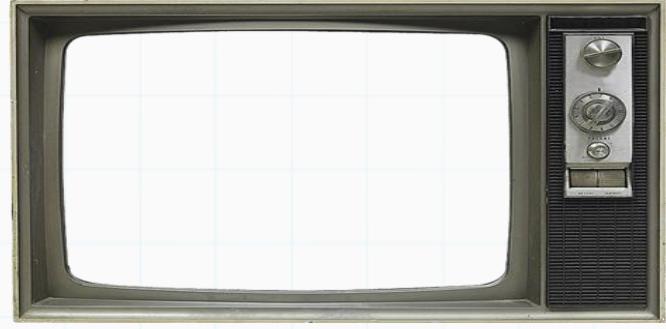
    }while(no != L);

    return tam;
}
```

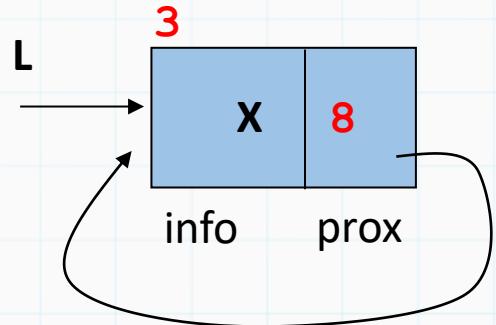


# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}



Se L vazio é fácil



```
// novo elemento
lista *novo;
novo      = aloca_no();
novo->info = el;
novo->prox = novo;

// Lista vazia
if (L == NULL)
    return novo;
```

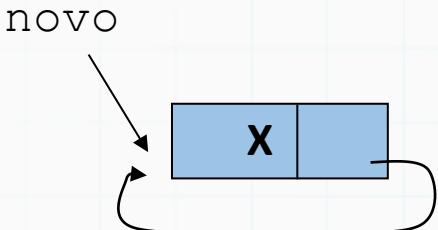
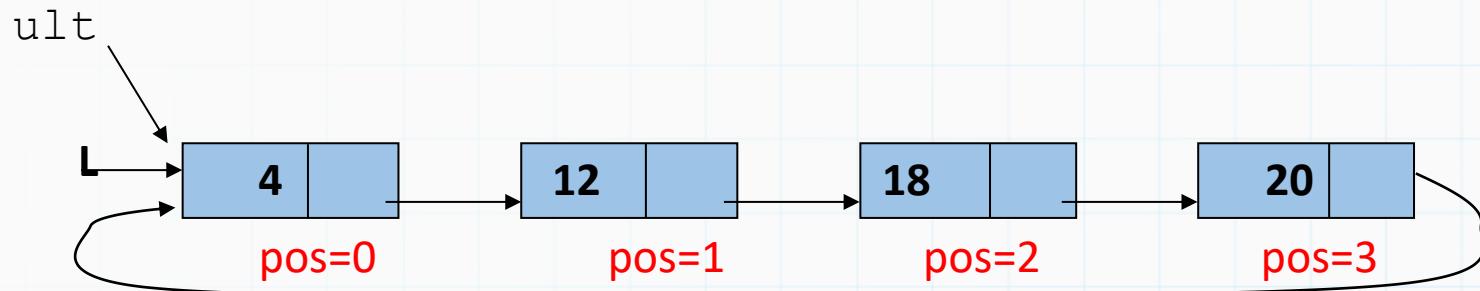
```
lista * aloca_no(void)
{
    lista *aux;
    aux      = (lista *) malloc (sizeof(lista));
    aux->prox = NULL;

    return aux;
}
```

# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $pos=0$ , então o elemento vai ser a nova cabeça da lista, vamos achar o ultimo elemento



```
if (pos==0)
{
    lista *ultimo = L;

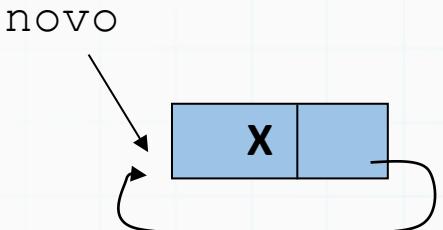
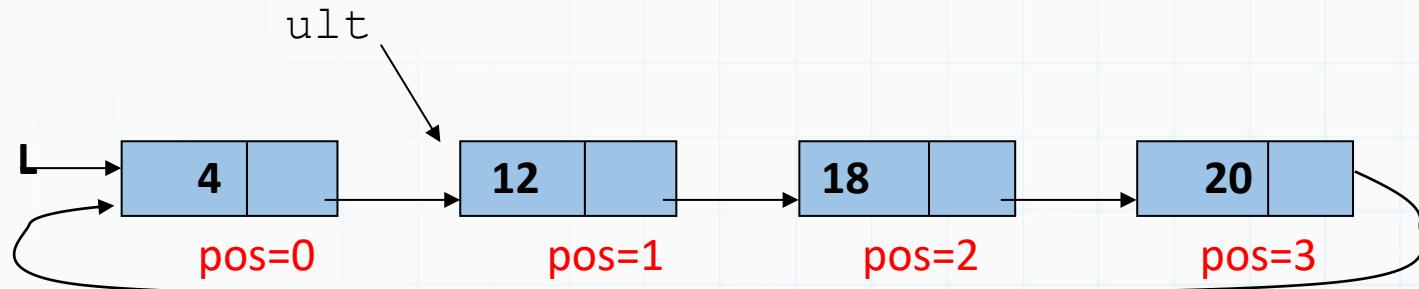
    do ultimo = ultimo->prox;
    while (ultimo->prox != L);

    ultimo->prox = novo;
    novo->prox = L;
    return novo;
}
```

# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $pos=0$ , então o elemento vai ser a nova cabeça da lista, vamos achar o ultimo elemento



```
if (pos==0)
{
    lista *ultimo = L;

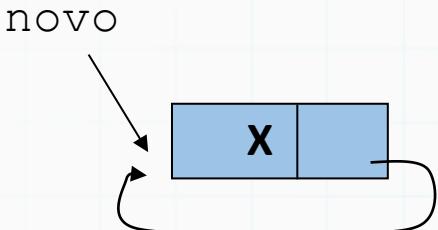
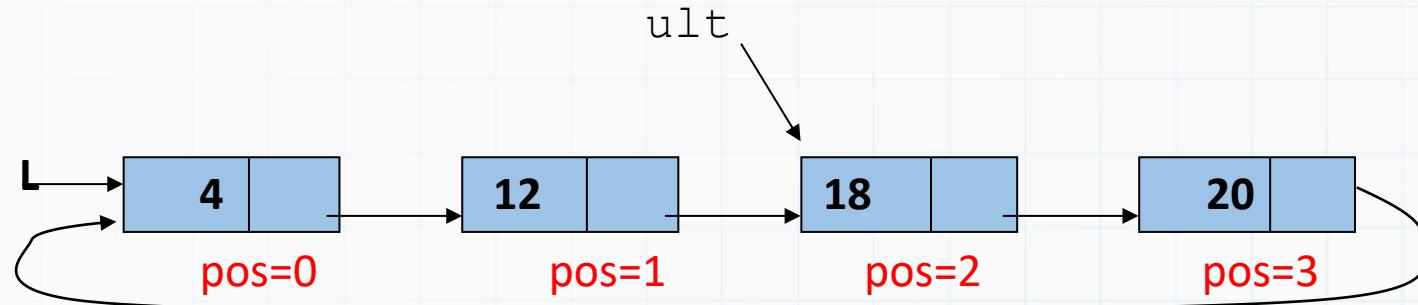
    do ultimo = ultimo->prox;
    while (ultimo->prox != L);

    ultimo->prox = novo;
    novo->prox = L;
    return novo;
}
```

# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $pos=0$ , então o elemento vai ser a nova cabeça da lista, vamos achar o ultimo elemento



```
if (pos==0)
{
    lista *ultimo = L;

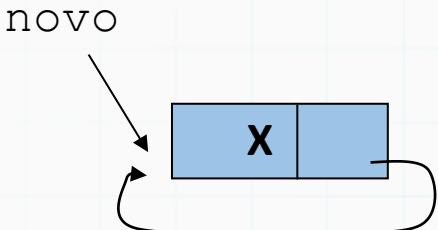
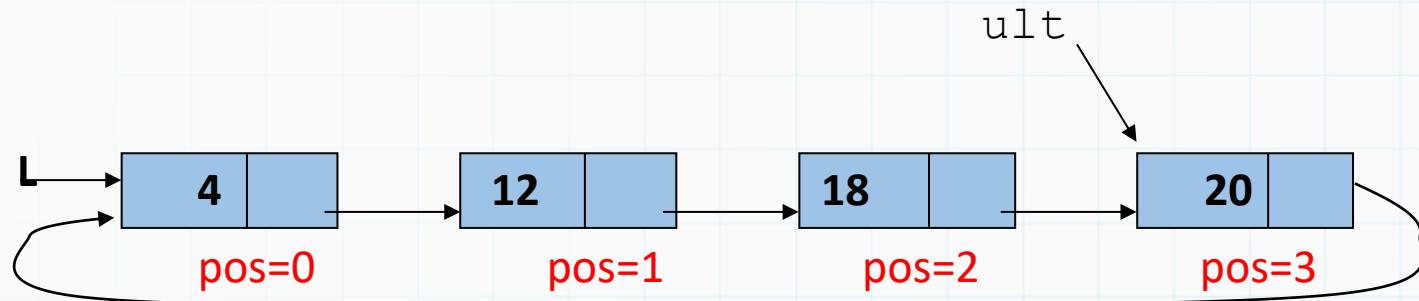
    do ultimo = ultimo->prox;
    while (ultimo->prox != L);

    ultimo->prox = novo;
    novo->prox = L;
    return novo;
}
```

# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $pos=0$ , então o elemento vai ser a nova cabeça da lista, vamos achar o ultimo elemento



```
if (pos==0)
{
    lista *ultimo = L;

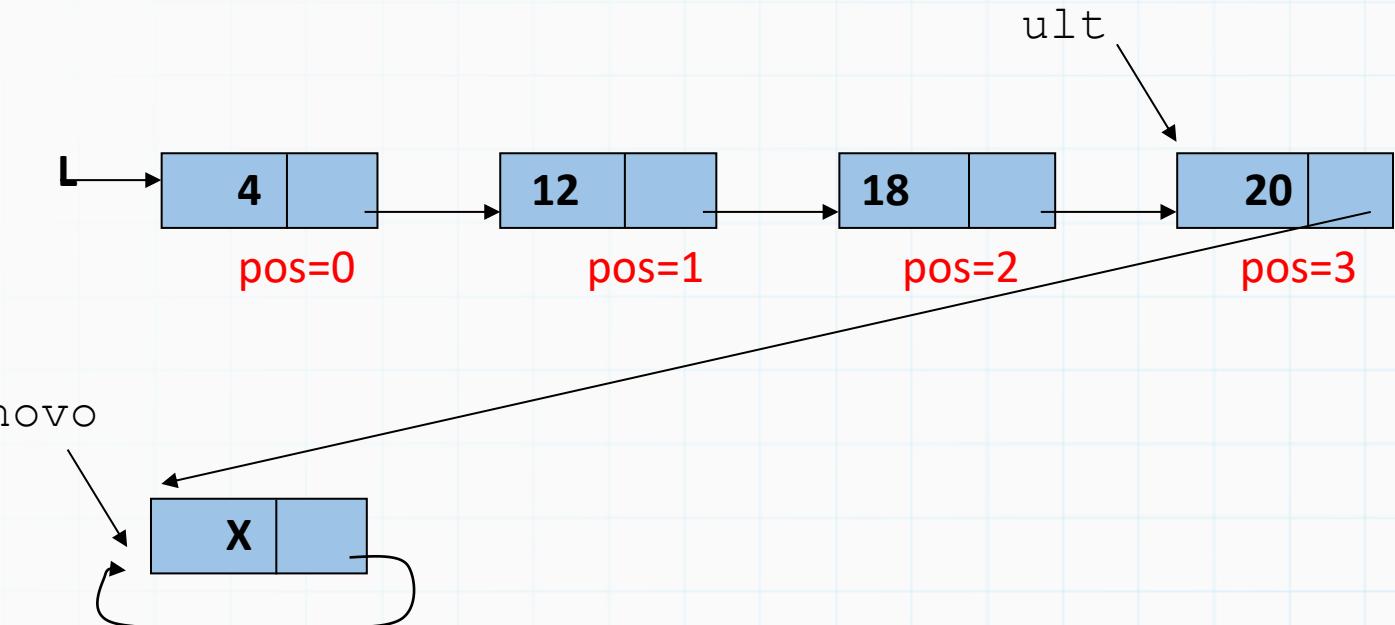
    do ultimo = ultimo->prox;
    while (ultimo->prox != L);

    ultimo->prox = novo;
    novo->prox = L;
    return novo;
}
```

# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $pos=0$ , então o elemento vai ser a nova cabeça da lista, vamos achar o ultimo elemento



```
if (pos==0)
{
    lista *ultimo = L;

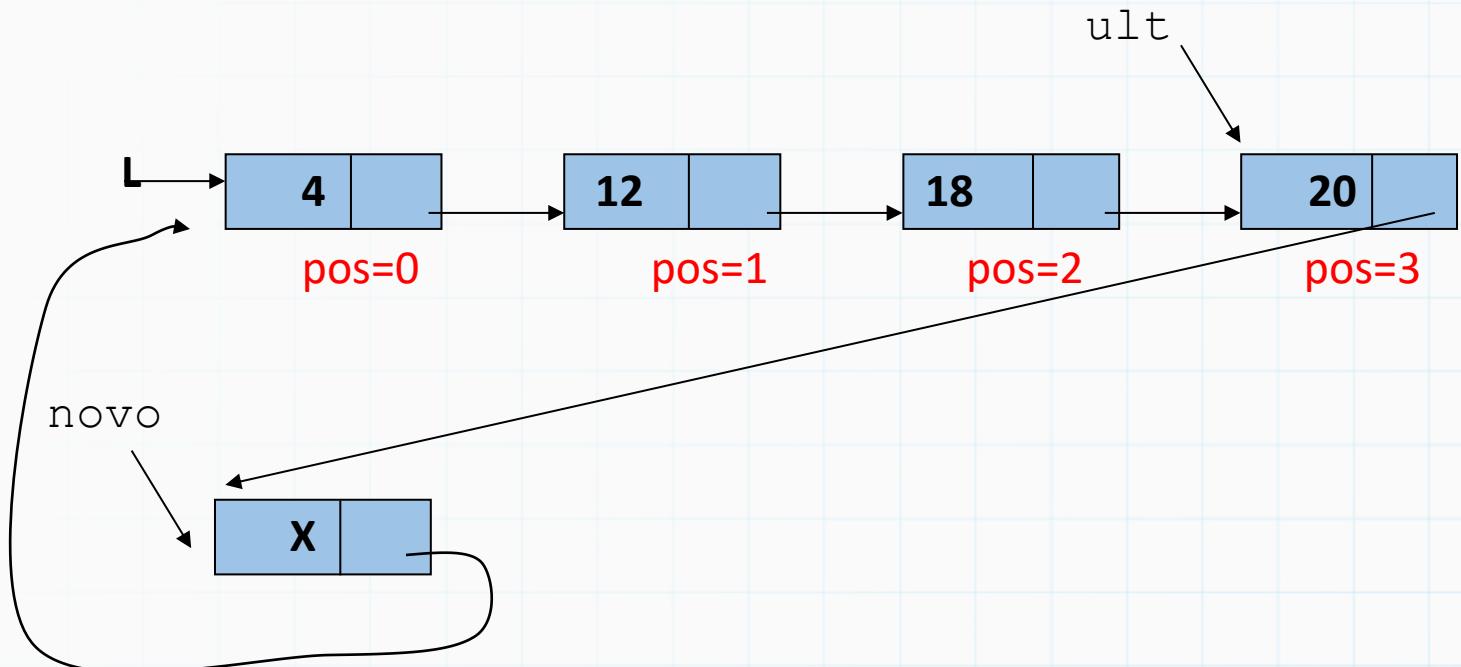
    do ultimo = ultimo->prox;
    while (ultimo->prox != L);

    ultimo->prox = novo;
    novo->prox = L;
    return novo;
}
```

# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $pos=0$ , então o elemento vai ser a nova cabeça da lista, vamos achar o ultimo elemento



```
if (pos==0)
{
    lista *ultimo = L;

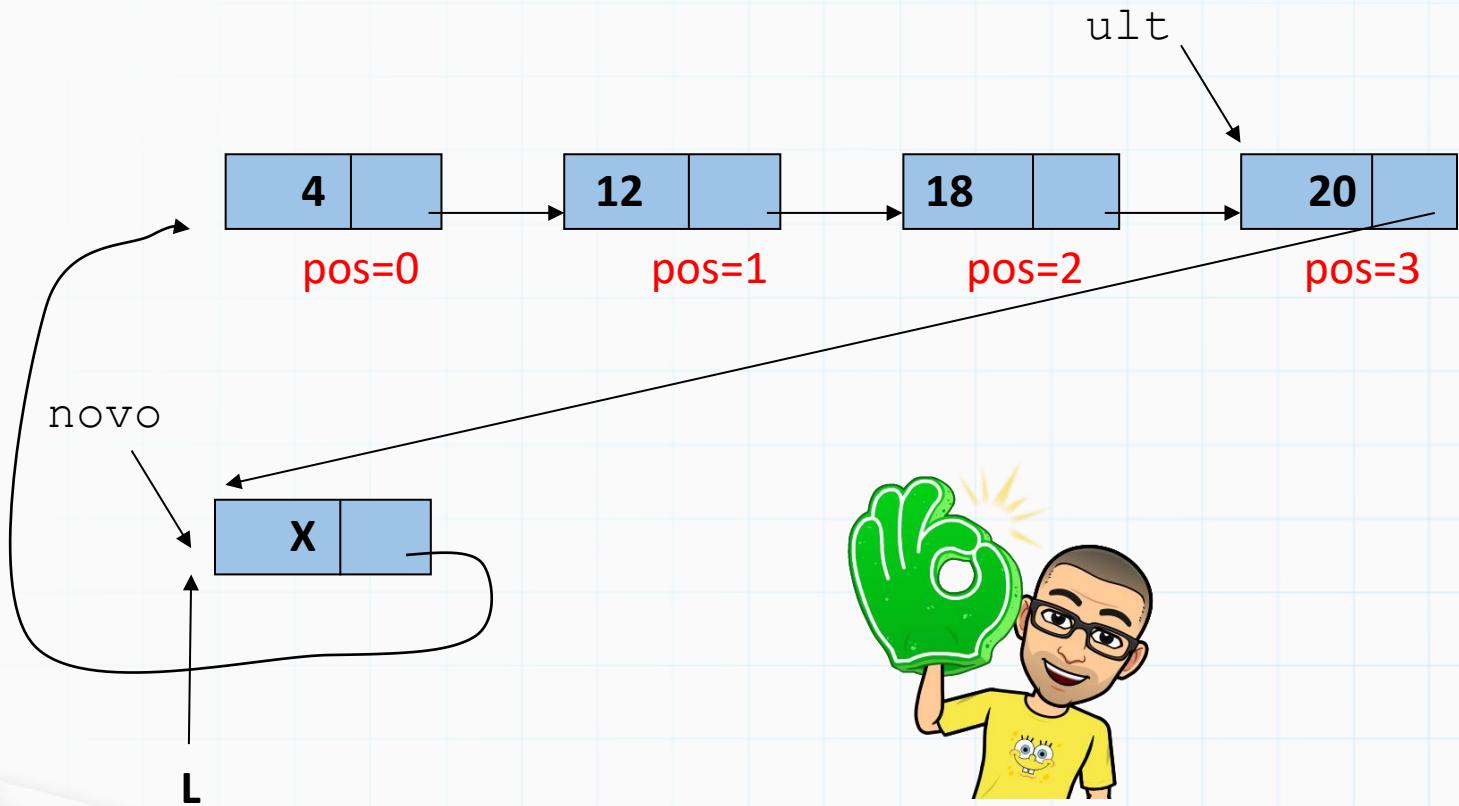
    do ultimo = ultimo->prox;
    while (ultimo->prox != L);

    ultimo->prox = novo;
novo->prox = L;
    return novo;
}
```

# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $pos=0$ , então o elemento vai ser a nova cabeça da lista, vamos achar o ultimo elemento



```
if (pos==0)
{
    lista *ultimo = L;

    do ultimo = ultimo->prox;
    while (ultimo->prox != L);

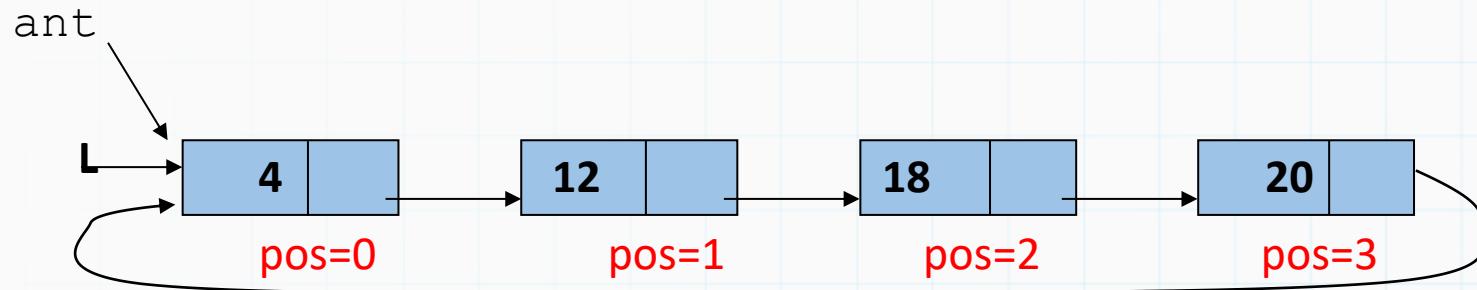
    ultimo->prox = novo;
    novo->prox = L;
    return novo;
}
```



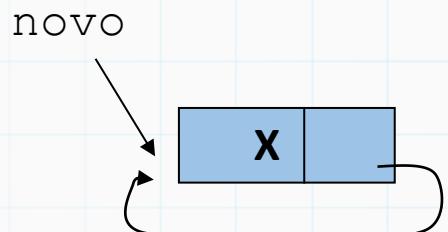
# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $0 < pos \leq n$ , por exemplo,  $pos=3$



Igual a lista não circular

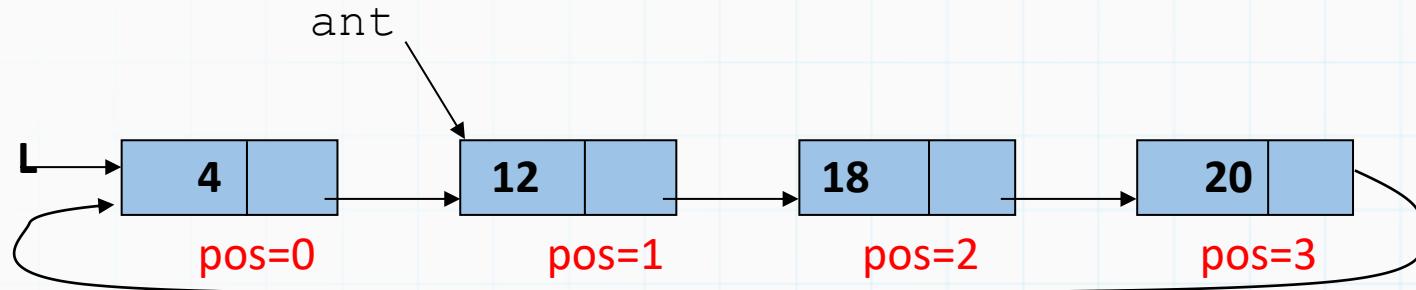


```
lista *anterior = L;  
for(int i=0; i<pos-1; i++)  
    anterior = anterior->prox;  
  
novo->prox      = anterior->prox;  
anterior->prox  = novo;  
  
return L;
```

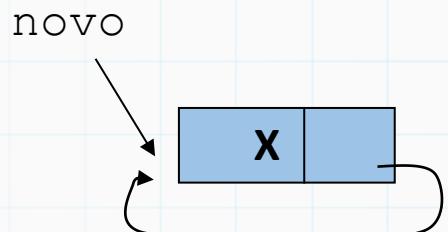
# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $0 < pos \leq n$ , por exemplo,  $pos=3$



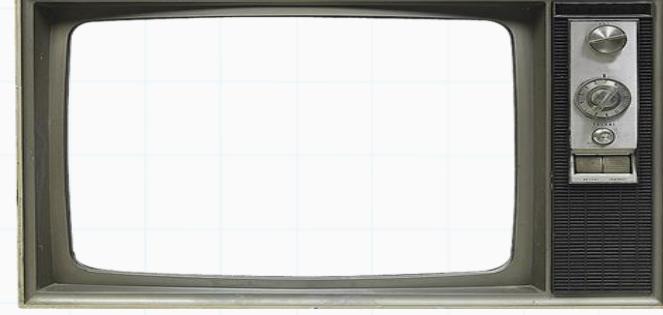
$i=0$



```
lista *anterior = L;
for(int i=0; i<pos-1; i++)
    anterior = anterior->prox;

novo->prox      = anterior->prox;
anterior->prox  = novo;

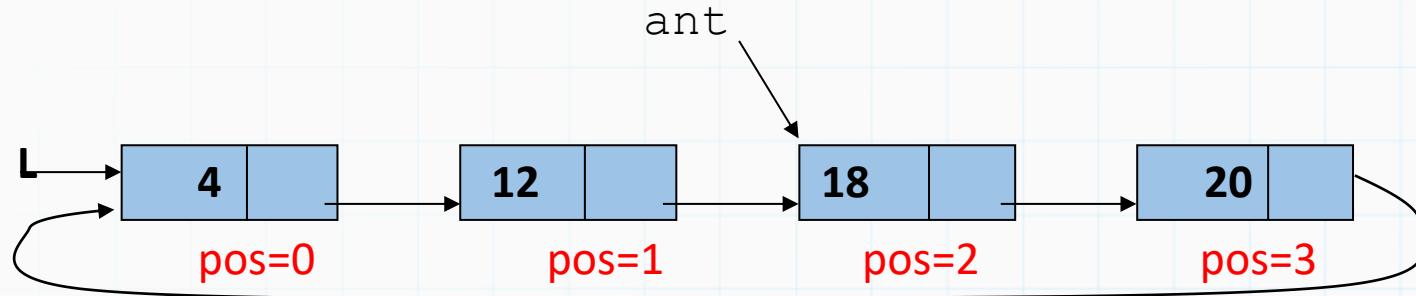
return L;
```



# Listas Circulares

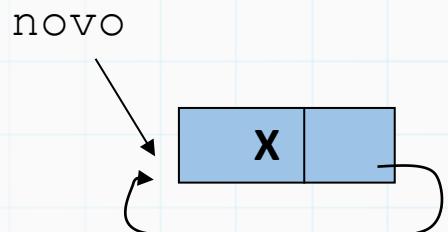
Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $0 < pos \leq n$ , por exemplo,  $pos=3$



Igual a lista não circular

$i=1$

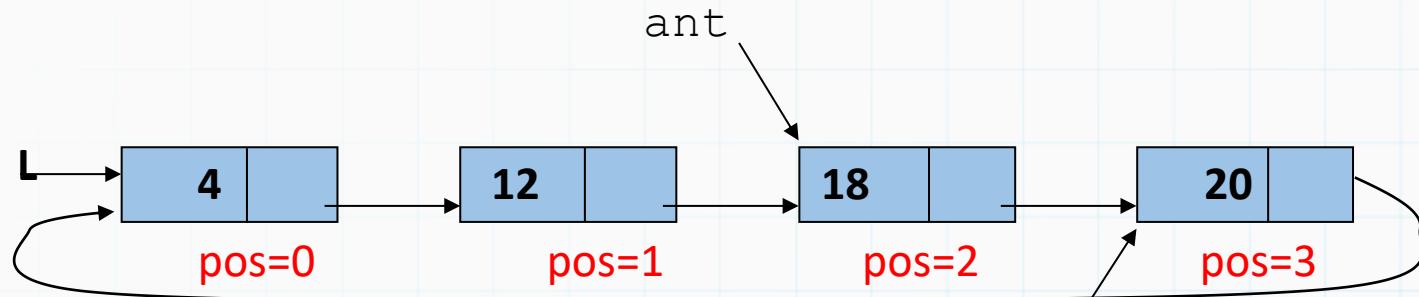


```
lista *anterior = L;  
for(int i=0; i<pos-1; i++)  
    anterior = anterior->prox;  
  
novo->prox      = anterior->prox;  
anterior->prox  = novo;  
  
return L;
```

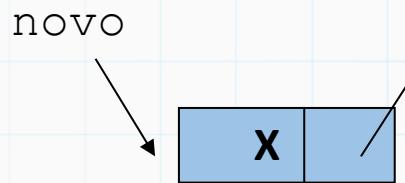
# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $0 < pos \leq n$ , por exemplo,  $pos=3$



$i=1$



```
lista *anterior = L;
for(int i=0; i<pos-1; i++)
    anterior = anterior->prox;

novo->prox      = anterior->prox;
anterior->prox = novo;

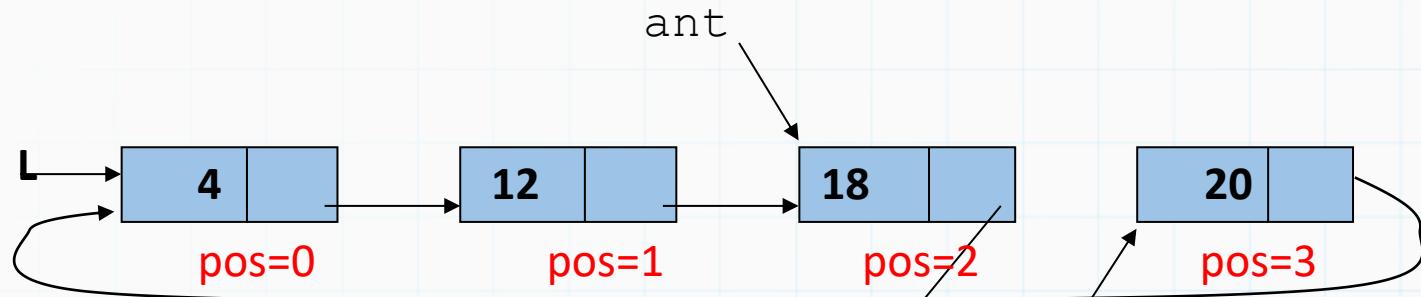
return L;
```

Igual a lista não circular

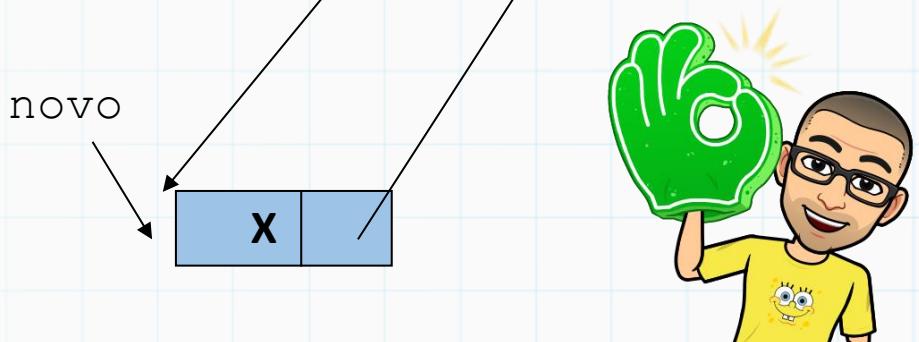
# Listas Circulares

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

se  $0 < pos \leq n$ , por exemplo,  $pos=3$



$i=1$



Igual a lista não circular

```
lista *anterior = L;
for(int i=0; i<pos-1; i++)
    anterior = anterior->prox;

novo->prox      = anterior->prox;
anterior->prox = novo;

return L;
```

```
lista* insere_lista_C(lista* L, int el, int pos)
{
    if (pos<0 || pos > tamanho(L))
    {
        printf("posicao invalida\n");
        return L;
    }

    // novo elemento
    lista *novo;
    novo = aloca_no();
    novo->info = el;
    novo->prox = novo;

    // Lista vazia
    if (L == NULL)
        return novo;

    if (pos==0)
    {
        lista *ultimo = L;

        do ultimo = ultimo->prox;
        while (ultimo->prox != L);

        ultimo->prox = novo;
        novo->prox = L;
        return novo;
    }

    if (pos>0)
    {
        lista *anterior = L;
        for(int i=0; i<pos-1; i++)
            anterior = anterior->prox;

        novo->prox = anterior->prox;
        anterior->prox = novo;
    }
}
```

## Listas Circulares

```
else
{
    lista *anterior = L;
    for(int i=0; i<pos-1; i++)
        anterior = anterior->prox;

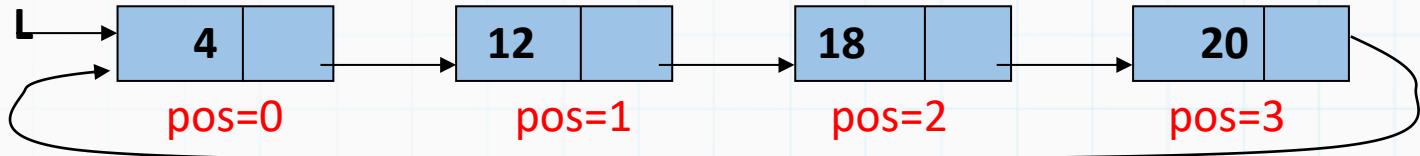
    novo->prox = anterior->prox;
    anterior->prox = novo;

    return L;
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



Temos que achar o ultimo no

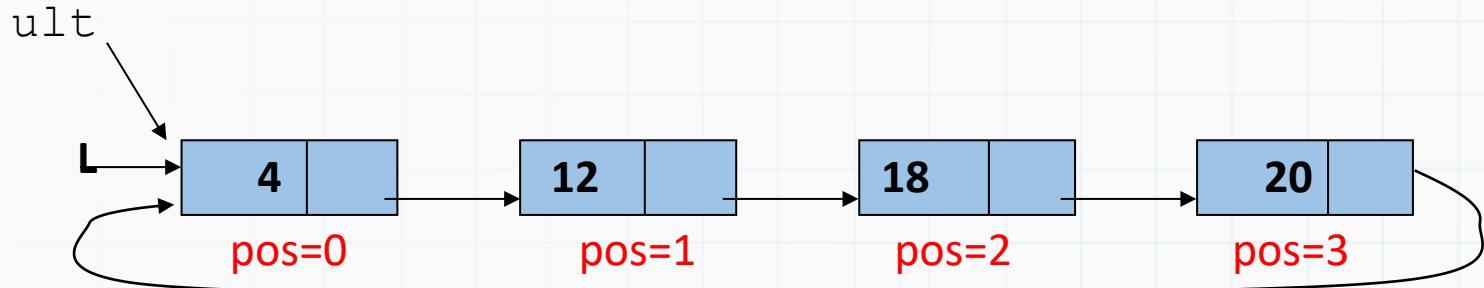
```
if(pos==0)
{
    lista* ultimo = L;
    while(ultimo->prox != L)
        ultimo = ultimo->prox;

    // so tem um elemento
    if(ultimo == L)
    {
        free(L);
        return NULL;
    }
    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



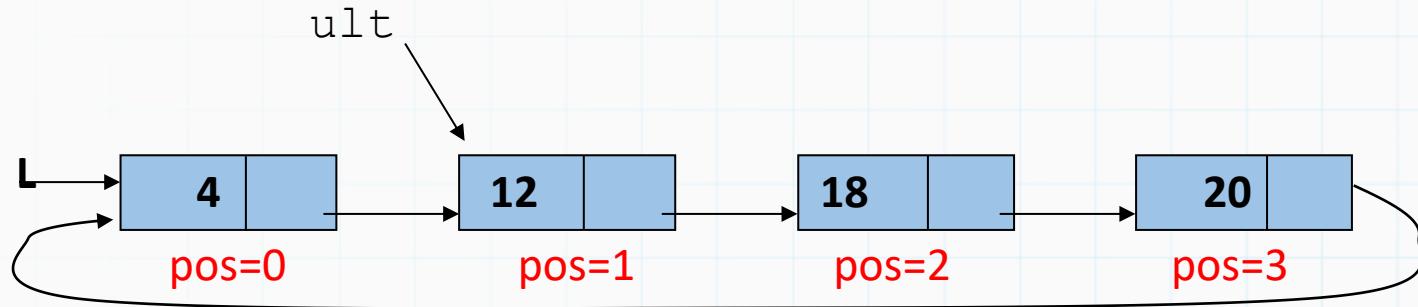
```
if(pos==0)
{
    lista* ultimo = L;
    while(ultimo->prox != L)
        ultimo = ultimo->prox;

    // so tem um elemento
    if(ultimo == L)
    {
        free(L);
        return NULL;
    }
    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



Temos que achar o ultimo no

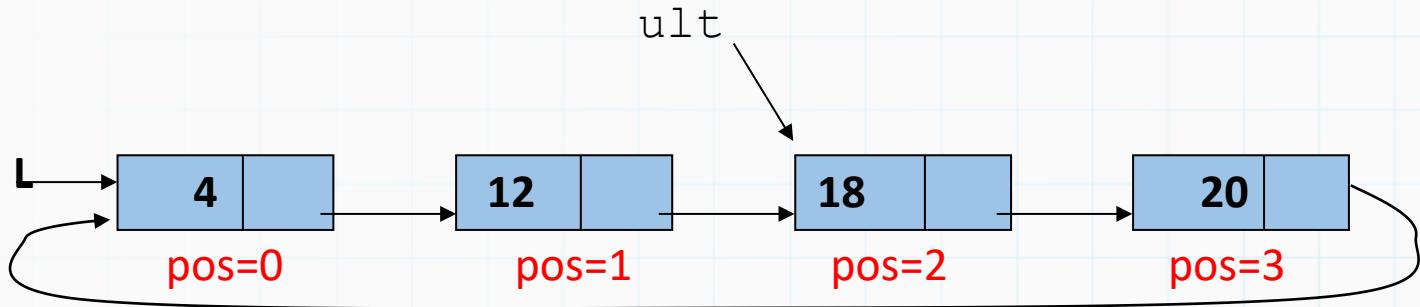
```
if(pos==0)
{
    lista* ultimo = L;
    while(ultimo->prox != L)
        ultimo = ultimo->prox;

    // so tem um elemento
    if(ultimo == L)
    {
        free(L);
        return NULL;
    }
    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



Temos que achar o ultimo no

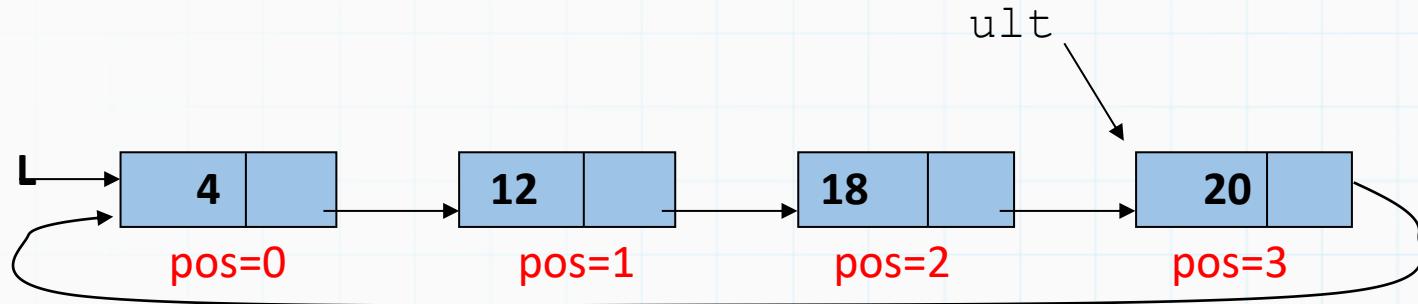
```
if(pos==0)
{
    lista* ultimo = L;
    while(ultimo->prox != L)
        ultimo = ultimo->prox;

    // so tem um elemento
    if(ultimo == L)
    {
        free(L);
        return NULL;
    }
    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



Temos que achar o ultimo no

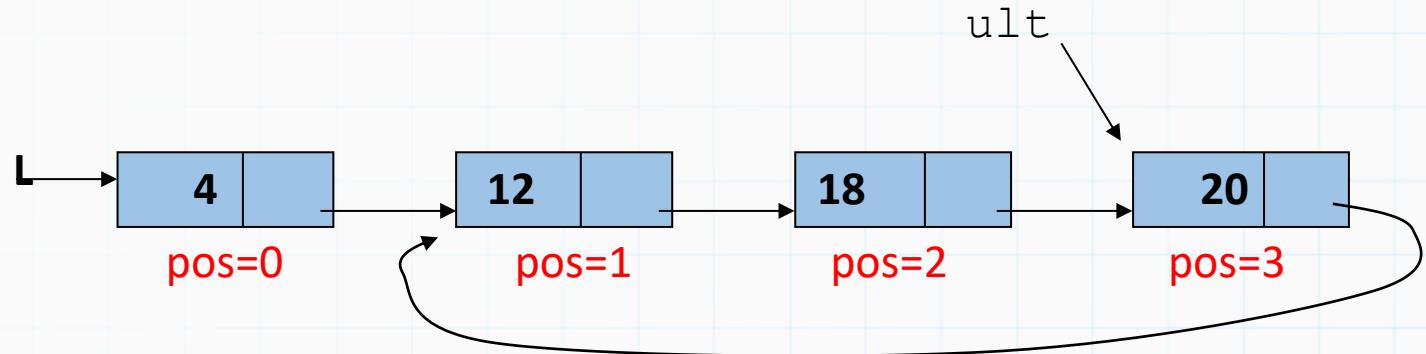
```
if(pos==0)
{
    lista* ultimo = L;
    while(ultimo->prox != L)
        ultimo = ultimo->prox;

    // so tem um elemento
    if(ultimo == L)
    {
        free(L);
        return NULL;
    }
    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



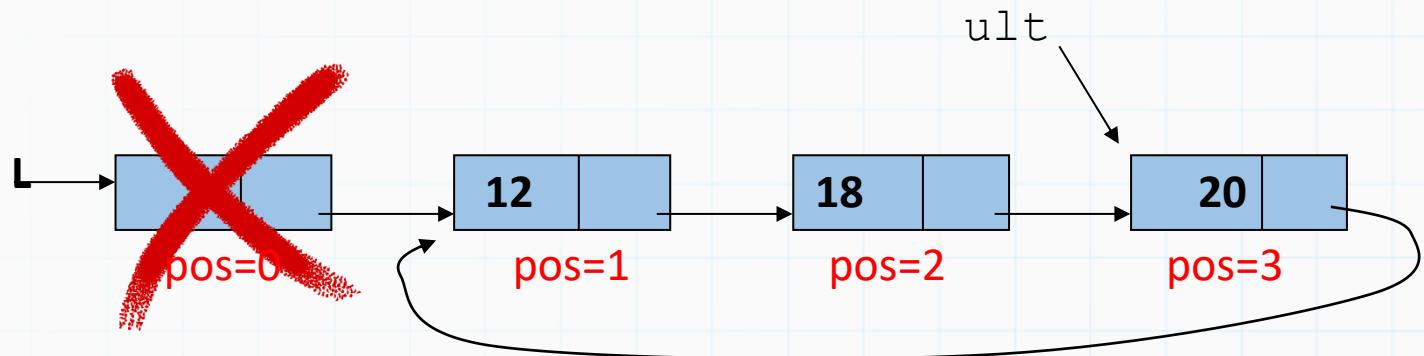
```
if(pos==0)
{
    lista* ultimo = L;
    while(ultimo->prox != L)
        ultimo = ultimo->prox;

    // so tem um elemento
    if(ultimo == L)
    {
        free(L);
        return NULL;
    }
    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



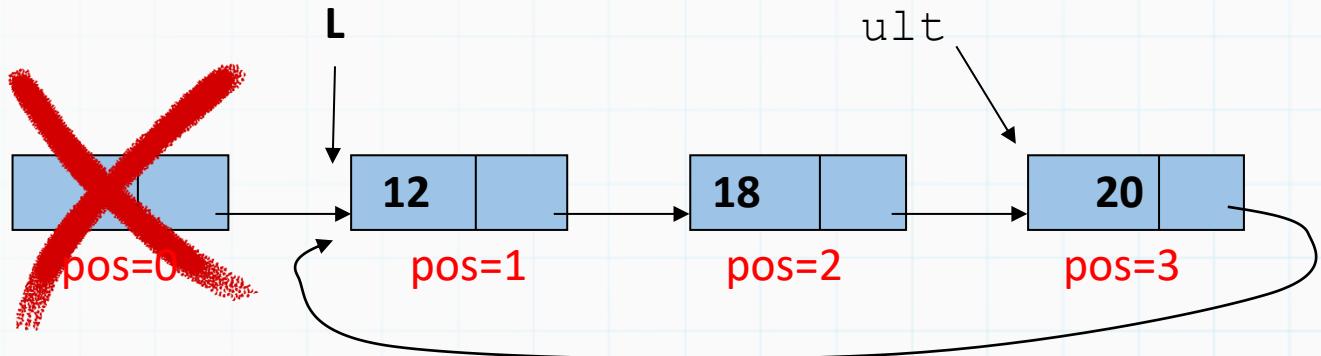
```
if(pos==0)
{
    lista* ultimo = L;
    while(ultimo->prox != L)
        ultimo = ultimo->prox;

    // so tem um elemento
    if(ultimo == L)
    {
        free(L);
        return NULL;
    }
    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



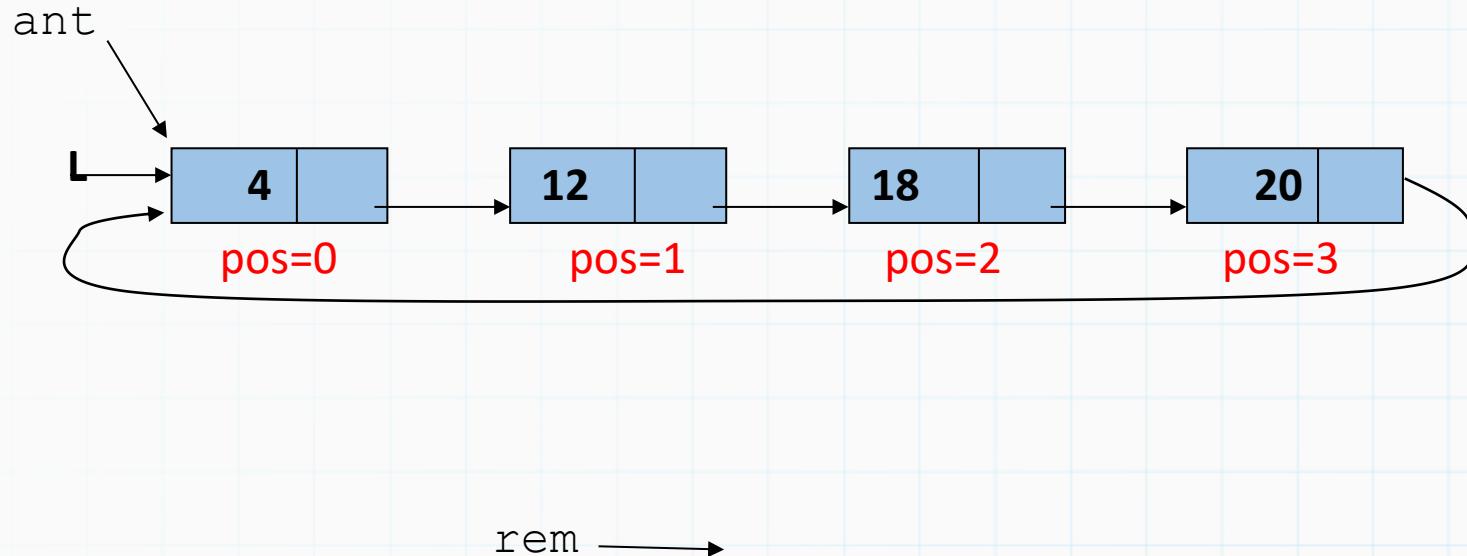
```
if(pos==0)
{
    lista* ultimo = L;
    while(ultimo->prox != L)
        ultimo = ultimo->prox;

    // so tem um elemento
    if(ultimo == L)
    {
        free(L);
        return NULL;
    }
    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
}
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$n-1 \geq pos > 0$ , por exemplo  $pos=2$



Igual a lista não circular

```
lista* anterior = L;
lista* removido;

for(int i=0; i < pos-1; i++)
    anterior = anterior->prox;

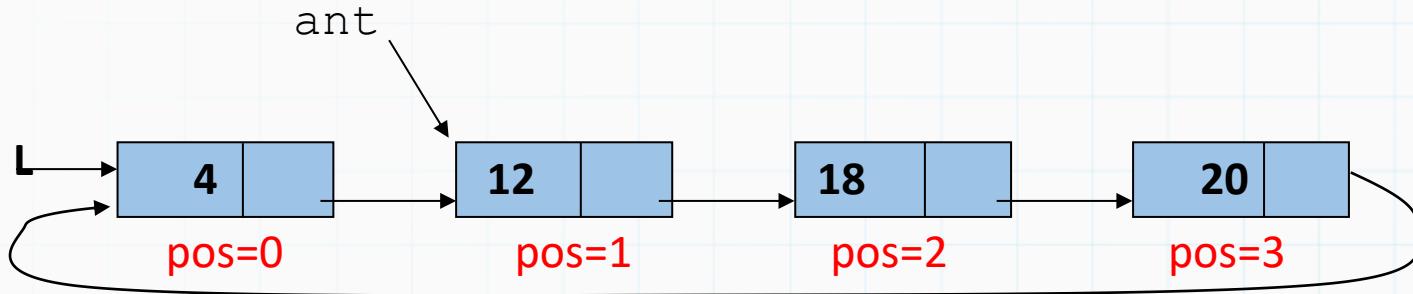
removido      = anterior->prox;
anterior->prox = removido->prox;

free(removido);
return L;
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$n-1 \geq pos > 0$ , por exemplo  $pos=2$



$i=0$

$rem \rightarrow$

Igual a lista não circular

```
lista* anterior = L;
lista* removido;

for(int i=0; i< pos-1; i++)
    anterior = anterior->prox;

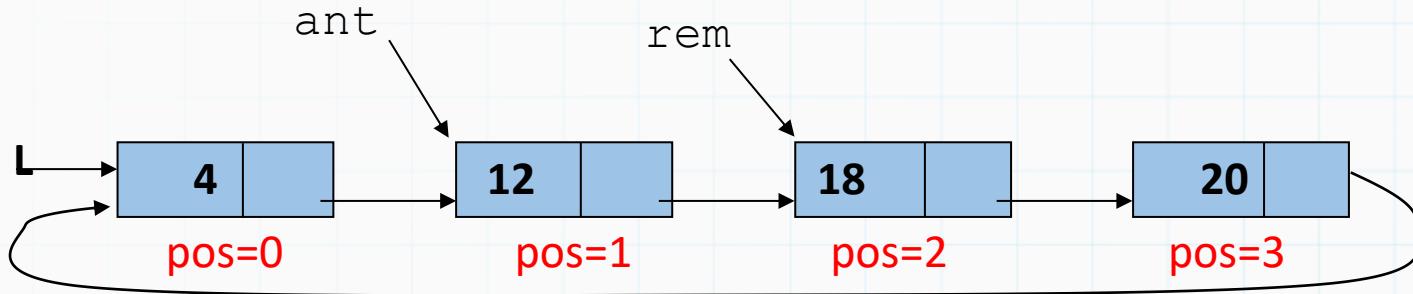
removido      = anterior->prox;
anterior->prox = removido->prox;

free(removido);
return L;
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$n-1 \geq pos > 0$ , por exemplo  $pos=2$



$i=0$

Igual a lista não circular

```
lista* anterior = L;
lista* removido;

for(int i=0; i < pos-1; i++)
    anterior = anterior->prox;

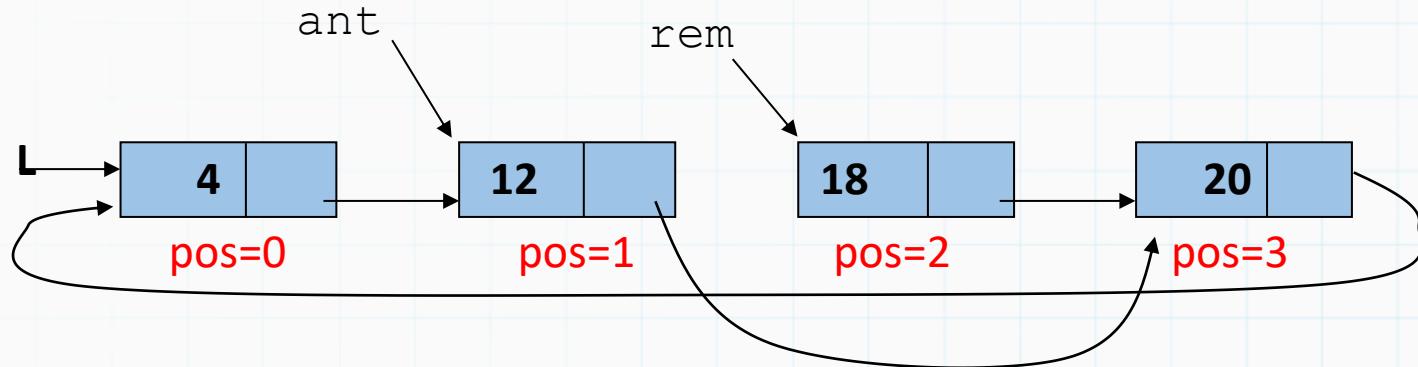
removido      = anterior->prox;
anterior->prox = removido->prox;

free(removido);
return L;
```

# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$n-1 \geq pos > 0$ , por exemplo  $pos=2$



$i=0$

Igual a lista não circular

```
lista* anterior = L;
lista* removido;

for(int i=0; i < pos-1; i++)
    anterior = anterior->prox;

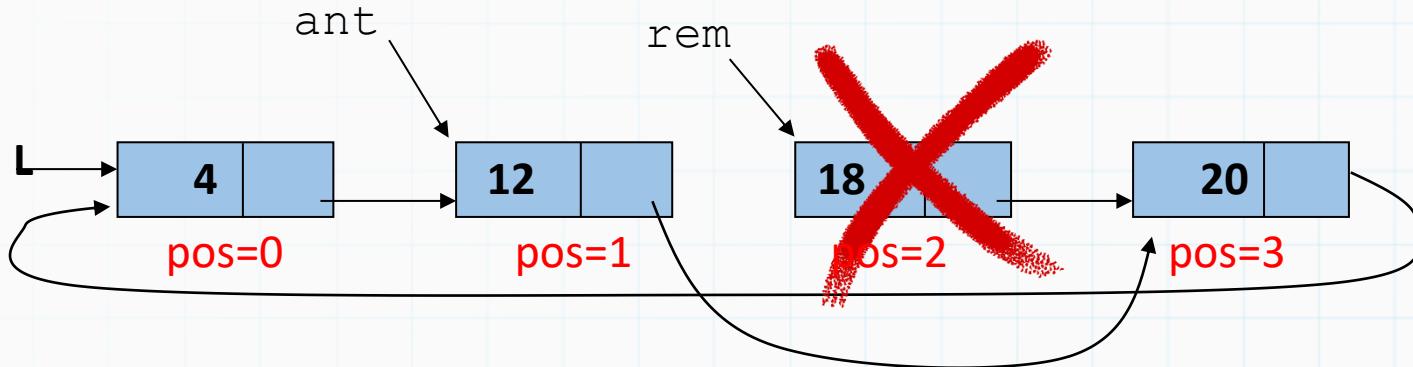
removido      = anterior->prox;
anterior->prox = removido->prox;

free(removido);
return L;
```

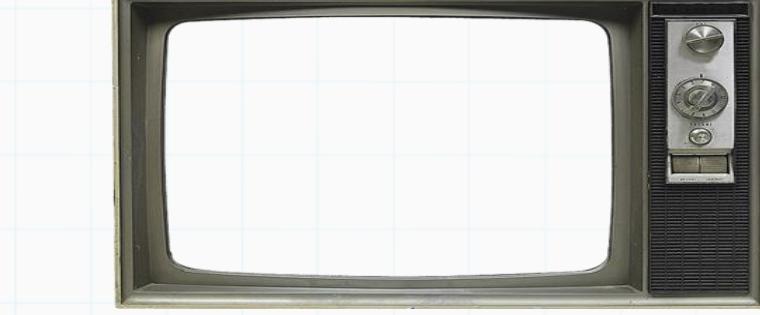
# Listas Circulares

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$n-1 \geq pos > 0$ , por exemplo  $pos=2$



$i=0$



Igual a lista não circular

```
lista* anterior = L;
lista* removido;

for(int i=0; i < pos-1; i++)
    anterior = anterior->prox;

removido      = anterior->prox;
anterior->prox = removido->prox;

free(removido);
return L;
```

```

lista* remove_lista_C(lista* L, int pos)
{
    // Lista vazia
    if (L == NULL)
        return L;

    if (pos<0 || pos > tamanho(L)-1)
    {
        printf("posicao invalida\n");
        return L;
    }

    if (pos==0)
    {
        lista* ultimo = L;
        while(ultimo->prox != L)
            ultimo = ultimo->prox;

        // se tem um elemento
        if(ultimo == L)
        {
            free(L);
            return NULL;
        }
    }

    // tem mais de um elemento
    else
    {
        ultimo->prox = L->prox;
        free(L);
        return ultimo->prox;
    }
    else
    {
        lista* anterior = L;
        lista* removido;

        for(int i=0; i< pos-1; i++)
            anterior = anterior->prox;

        removido = anterior->prox;
        anterior->prox = removido->prox;

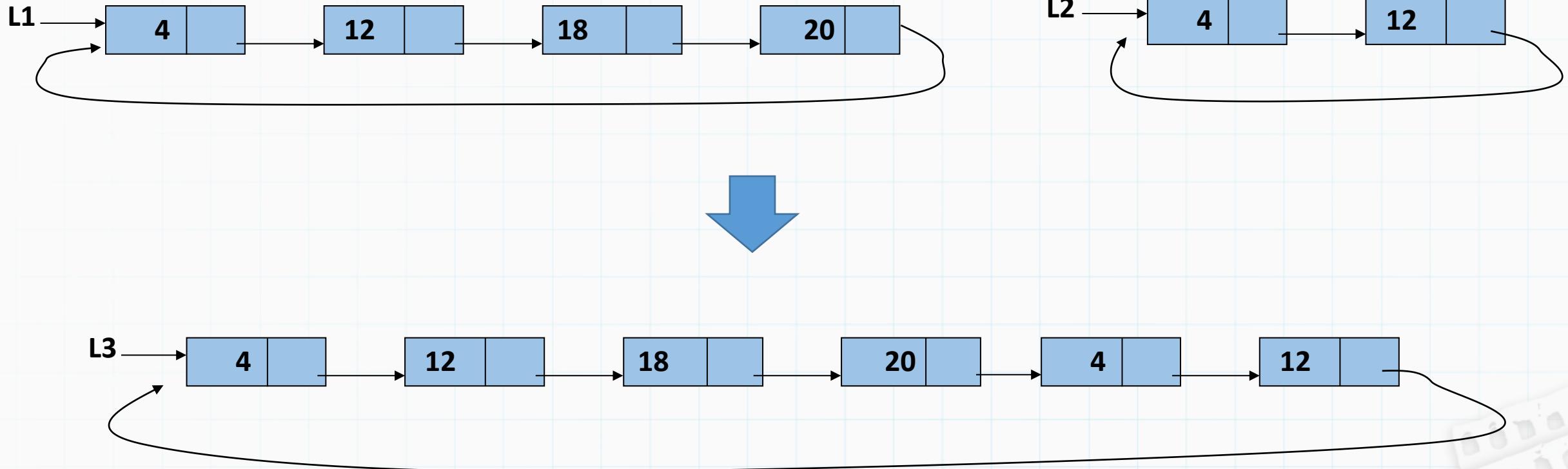
        free(removido);
        return L;
    }
}

```

# Listas Circulares



Exercício 1) Faça uma função que dado 2 listas circulares, concatene elas e retorne um ponteiro para a cabeça dessa lista nova.



Cuidado com os casos de listas vazias

```
struct NO {  
    int info;  
    struct NO *prox;  
}  
typedef struct NO lista;
```

# Listas encadeadas



Exercício 1) Faça uma função que dado 2 listas circulares, concatene elas e retorne um ponteiro para a cabeça dessa lista nova.

```
lista * concatena_listas_C (lista* L1, lista* L2)
{
    if (L1 == NULL)
        return L2;

    if (L2 == NULL)
        return L1;

    lista *ultimo = L1;
    do ultimo = ultimo->prox;
    while (ultimo->prox != L1);

    ultimo->prox = L2;

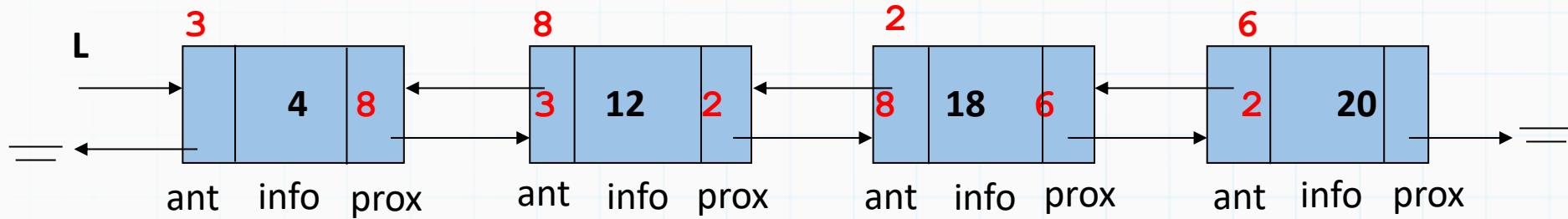
    ultimo = L2;
    do ultimo = ultimo->prox;
    while (ultimo->prox != L2);

    ultimo->prox = L1;

    return L1;
}
```

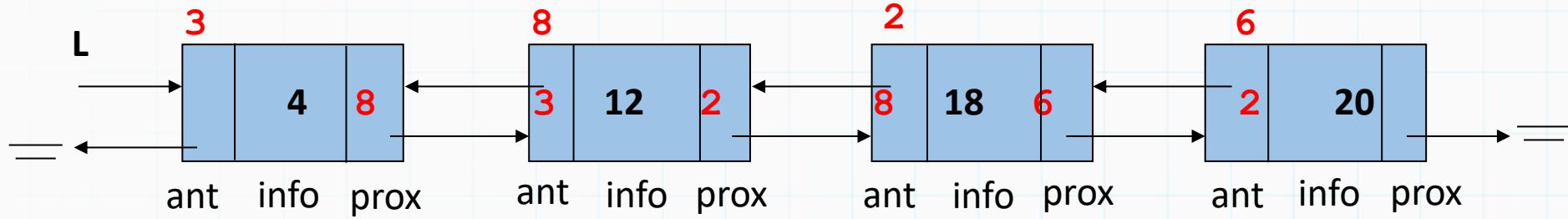
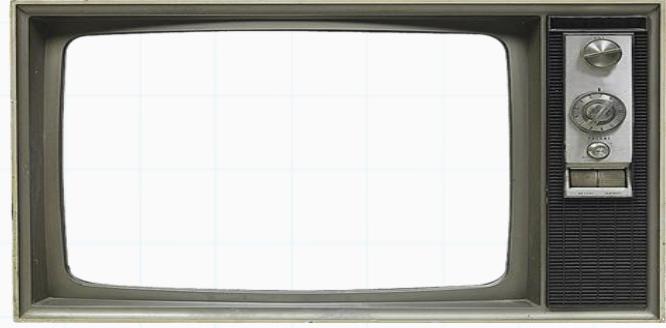
# Listas Duplamente Encadeadas

É uma lista duplamente encadeada, cada nó guarda o endereço do próximo e do seu antecessor.



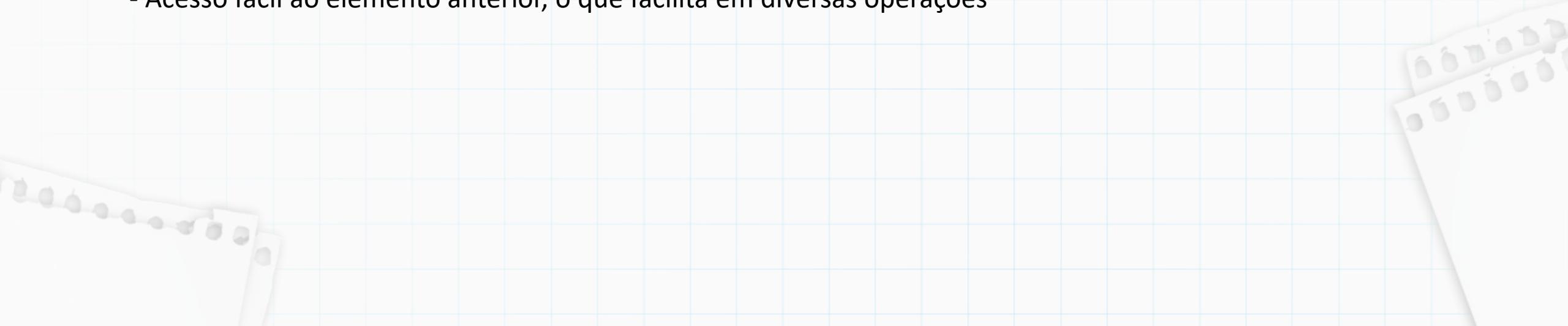
# Listas Duplamente Encadeadas

É uma lista duplamente encadeada, cada nó guarda o endereço do próximo e do seu antecessor.



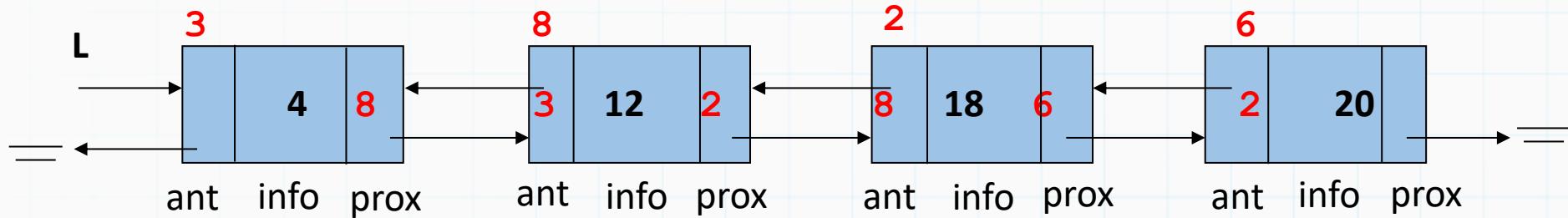
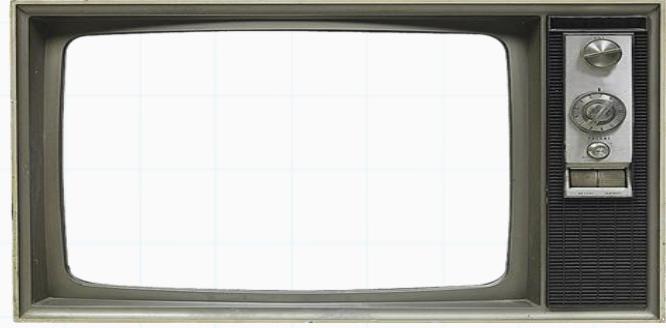
Vantagens:

- Acesso fácil ao elemento anterior, o que facilita em diversas operações



# Listas Duplamente Encadeadas

É uma lista duplamente encadeada, cada nó guarda o endereço do próximo e do seu antecessor.

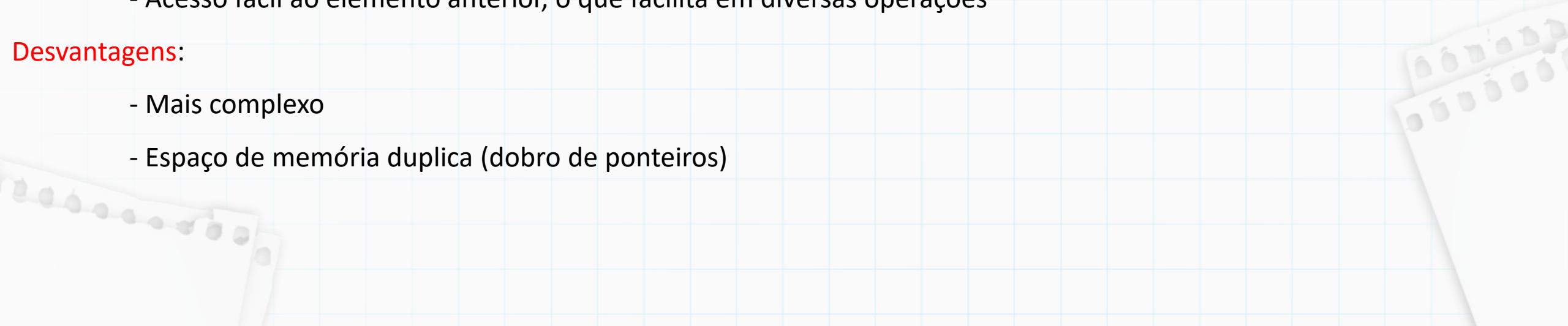


## Vantagens:

- Acesso fácil ao elemento anterior, o que facilita em diversas operações

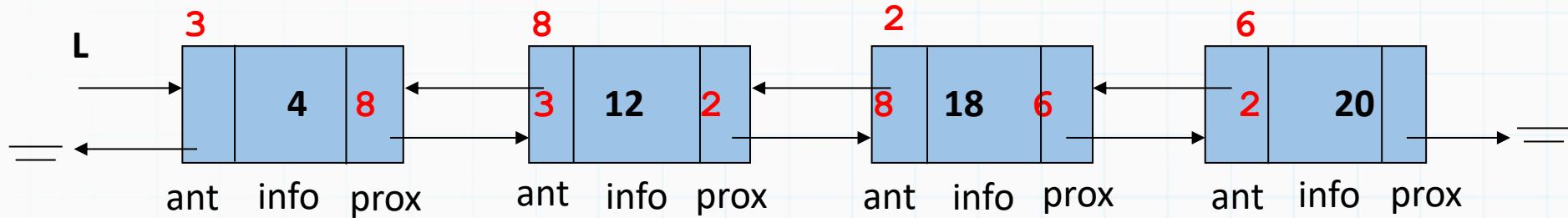
## Desvantagens:

- Mais complexo
- Espaço de memória duplica (dobro de ponteiros)



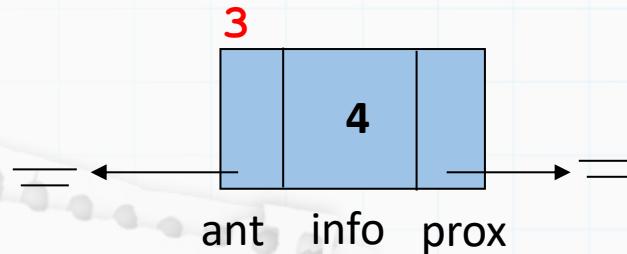
# Listas Duplamente Encadeadas

É uma lista duplamente encadeada, cada nó guarda o endereço do próximo e do seu antecessor.



Exemplos:

Um elemento



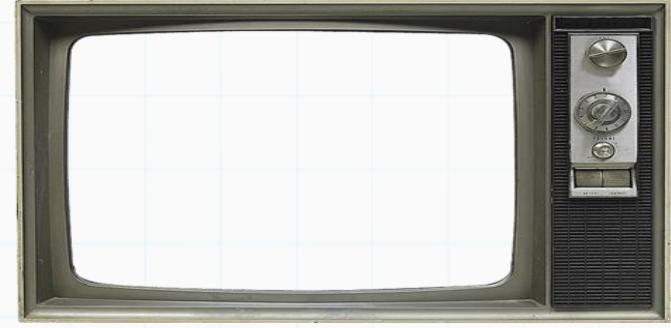
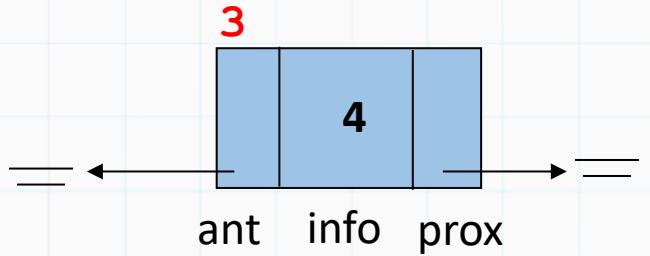
Vazia



# Listas Duplamente Encadeadas

- A estrutura de um nó da lista muda
- A inicialização também muda

```
struct NO {  
    int info;  
    struct NO *prox, *ant;  
}  
typedef struct NO lista;  
...  
lista *L;  
L = (lista*) malloc(sizeof(lista));  
L->prox = NULL;  
L->ant = NULL;
```



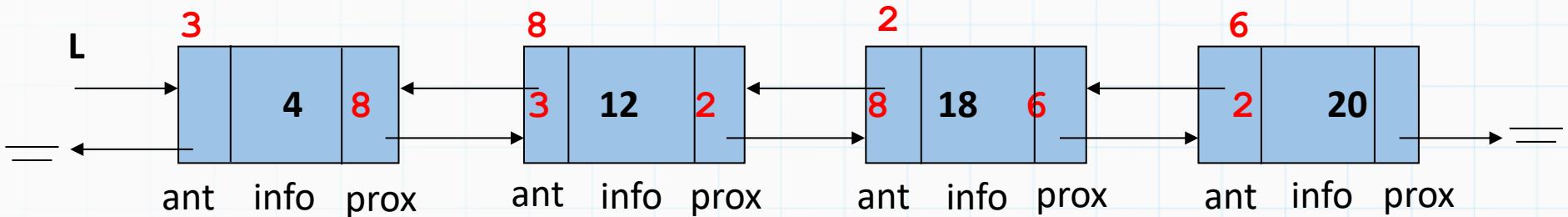
# Listas Duplamente Encadeadas

Percorrer: Igual a da lista encadeada

```
lista *no;
no = L;
while (no != NULL)
{
    printf("%d, ", no->info);
    no = no->prox;
}
```

```
struct NO {
    int info;
    struct NO *prox, *ant;
};

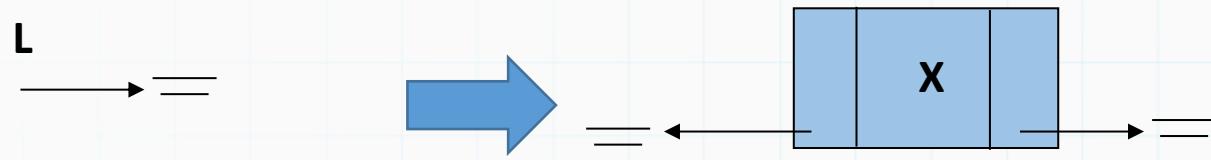
typedef struct NO lista;
```



# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

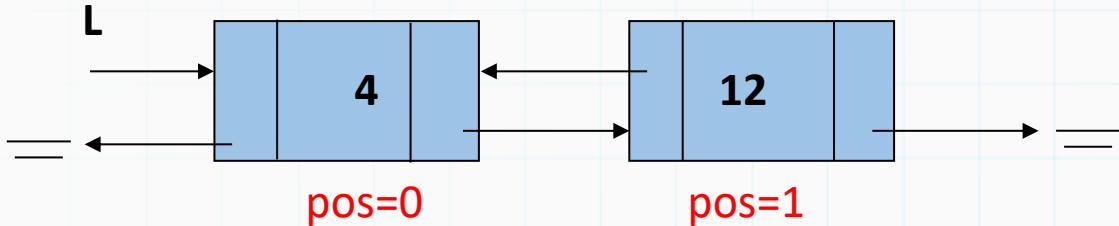
Se L vazio é fácil



# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se L vazio é fácil OU pos=0 é fácil



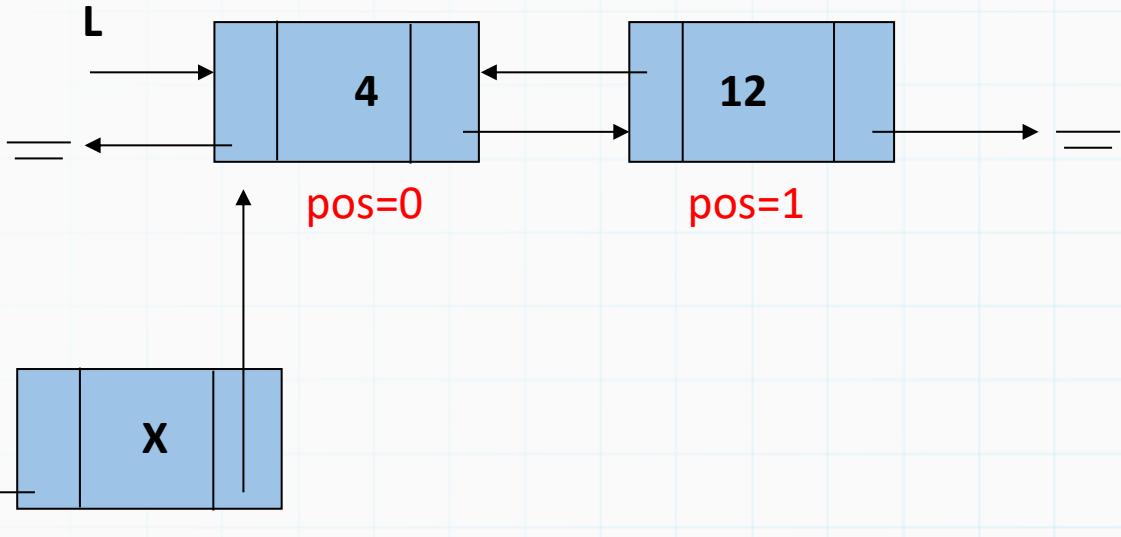
```
if ((pos == 0) || (L == NULL))
{
    lista *no;
    no      = alocar_no();
    no->info = el;
    no->prox = L;
    no->ant = NULL;

    if (L != NULL)
        L->ant = no;
    return no;
}
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se L vazio é fácil OU pos=0 é fácil



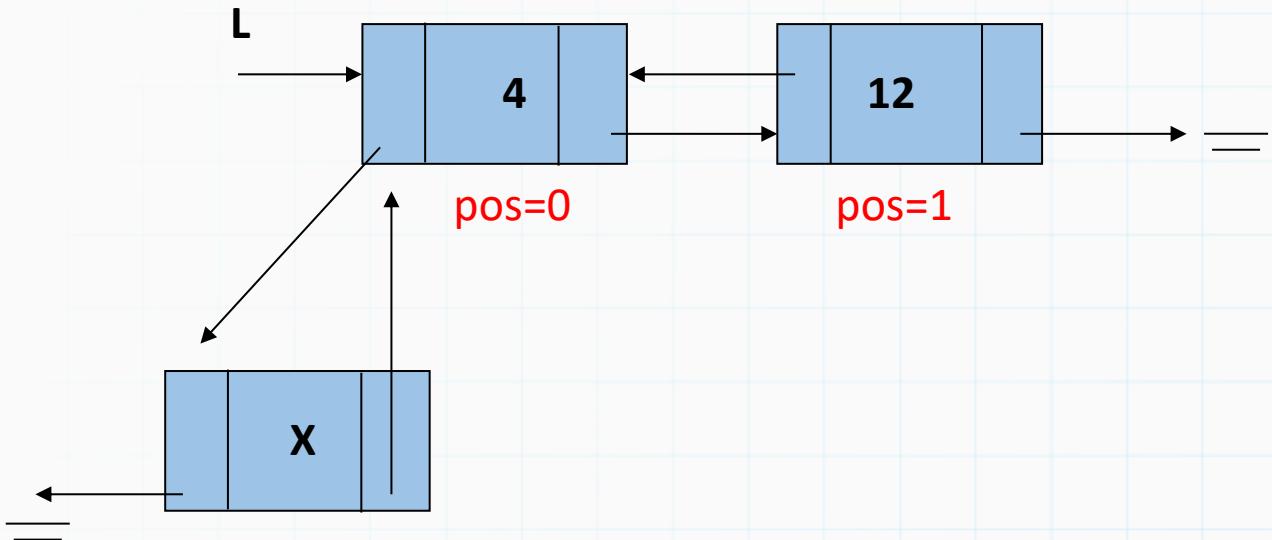
```
if ((pos == 0) || (L == NULL))
{
    lista *no;
    no      = aloca_no();
    no->info = el;
    no->prox = L;
    no->ant  = NULL;

    if (L != NULL)
        L->ant = no;
    return no;
}
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se L vazio é fácil OU pos=0 é fácil



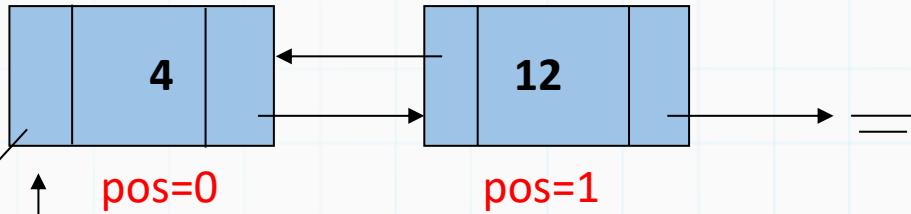
```
if ((pos == 0) || (L == NULL))
{
    lista *no;
    no      = aloca_no();
    no->info = el;
    no->prox = L;
    no->ant = NULL;

    if (L != NULL)
        L->ant = no;
    return no;
}
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se L vazio é fácil OU pos=0 é fácil



pos=0

pos=1

X

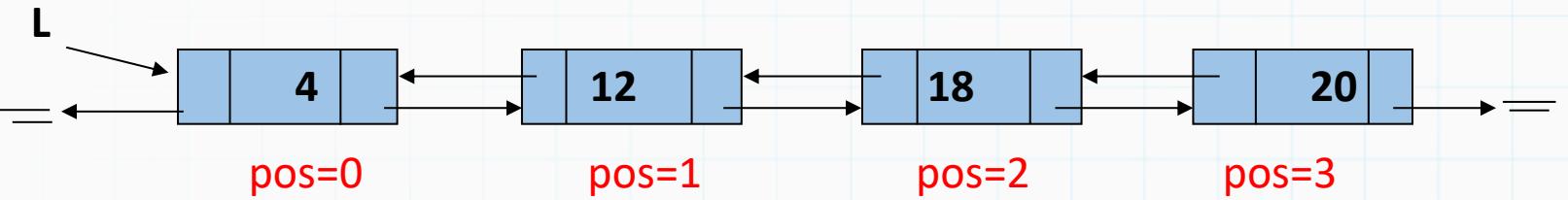


```
if ((pos == 0) || (L == NULL))  
{  
    lista *no;  
    no      = aloca_no();  
    no->info  = el;  
    no->prox  = L;  
    no->ant   = NULL;  
  
    if (L != NULL)  
        L->ant    = no;  
    return no;  
}
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

Se  $pos>0$ , por exemplo  $pos=2$

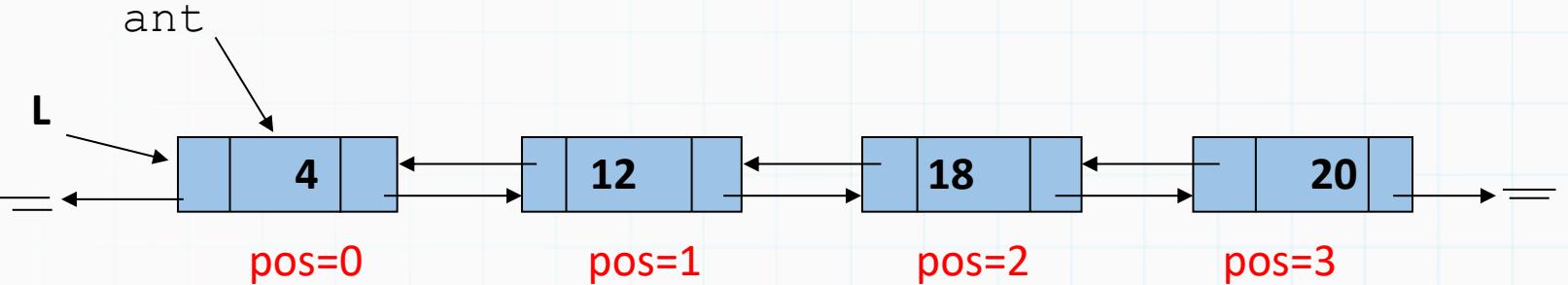


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se pos>0, por exemplo pos=2

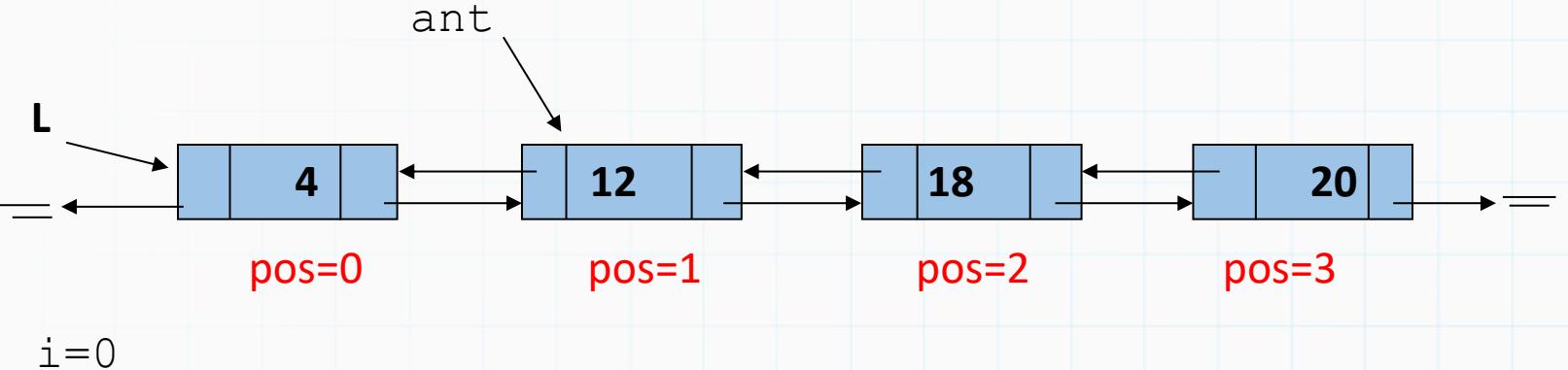


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

Se  $pos>0$ , por exemplo  $pos=2$

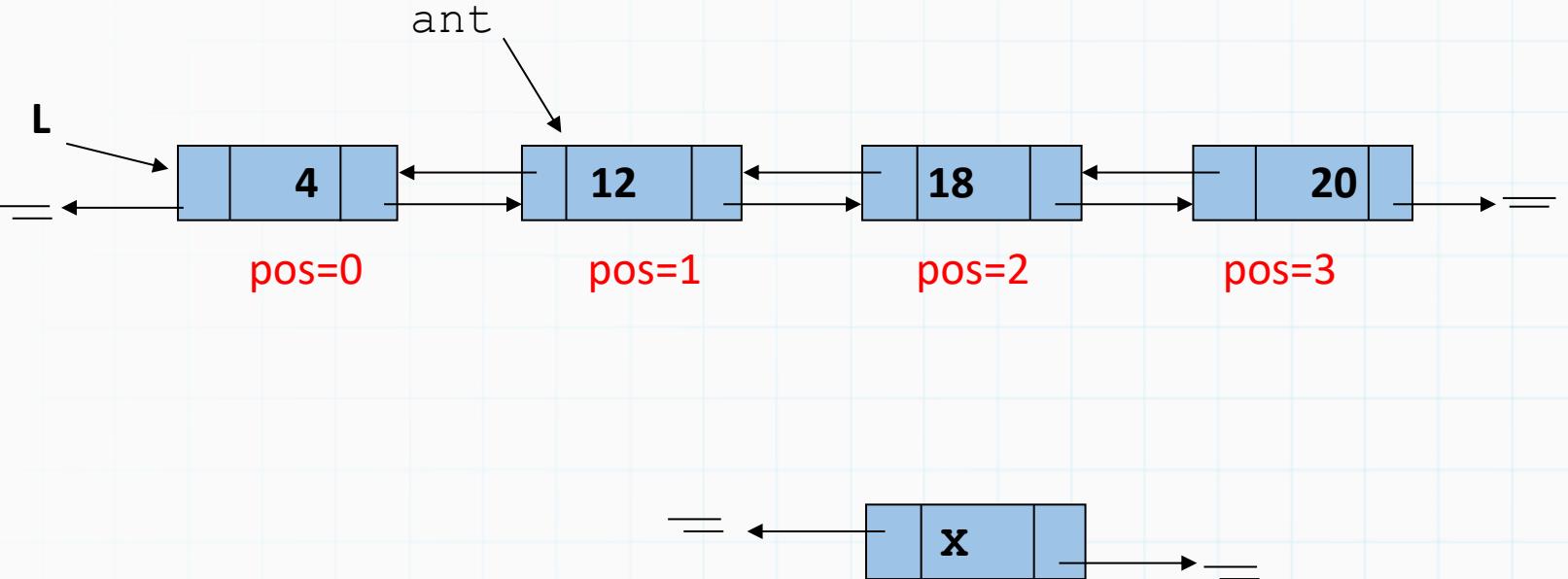


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se pos>0, por exemplo pos=2

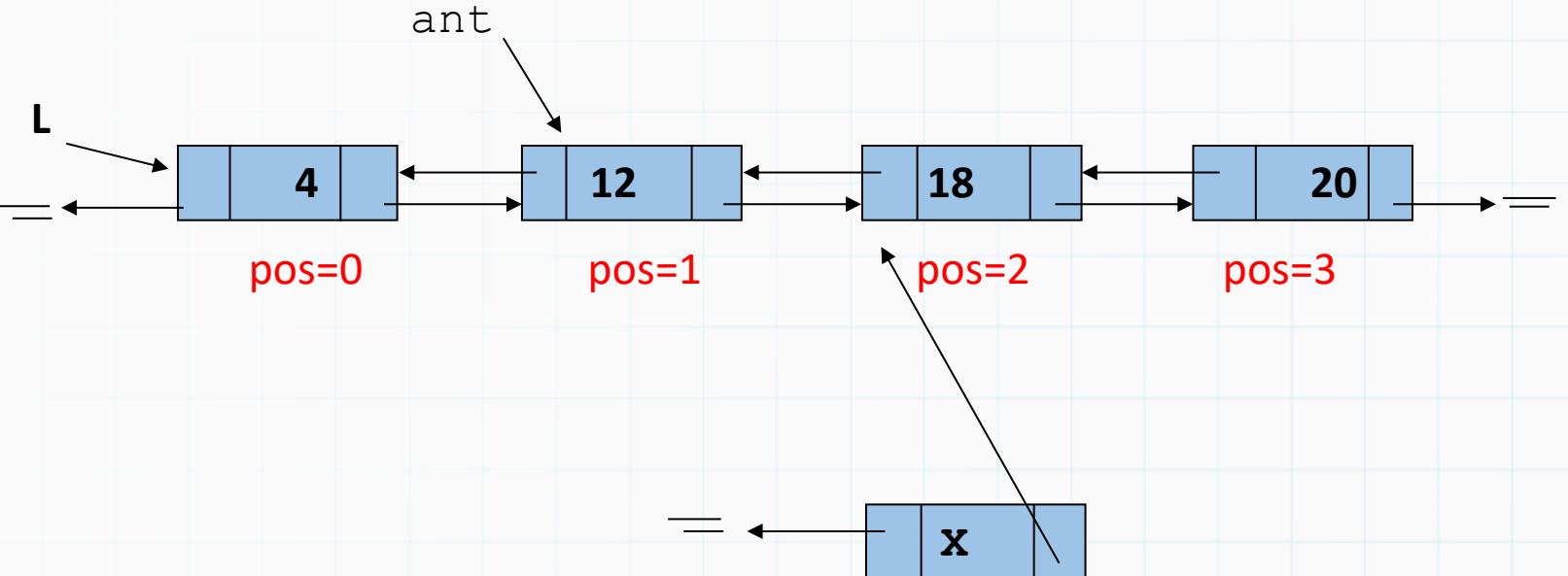


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant      = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se pos>0, por exemplo pos=2

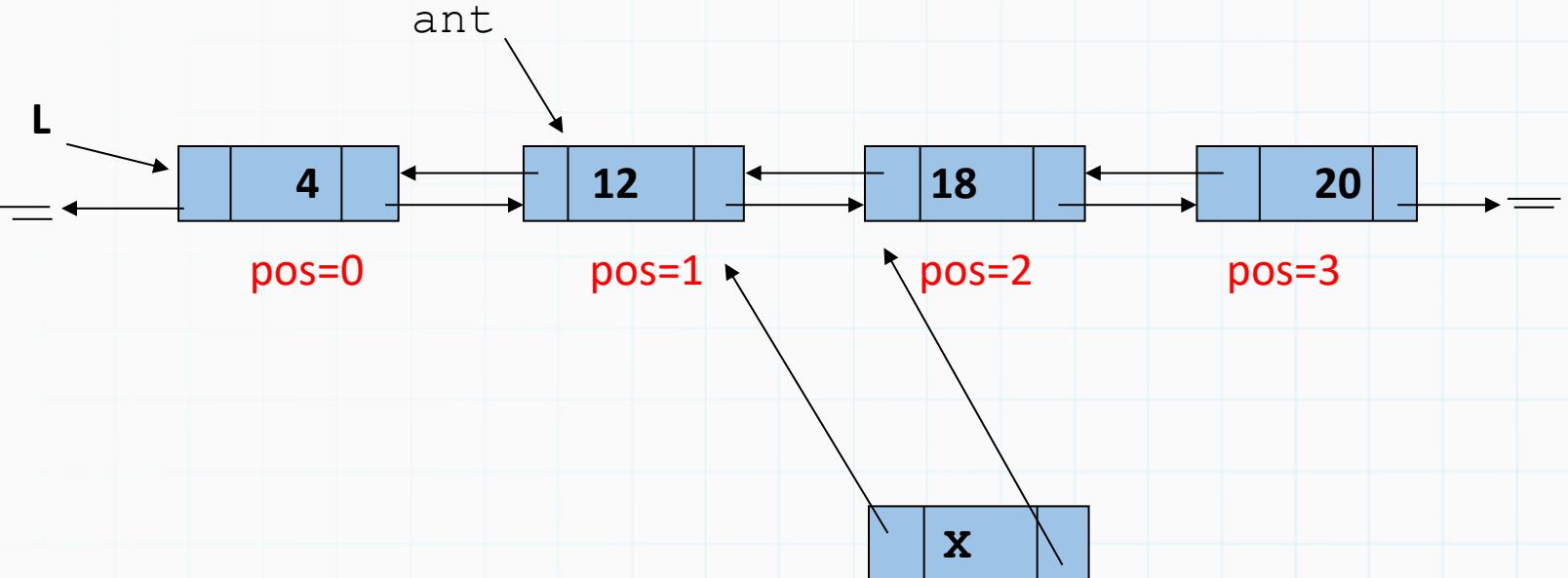


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

Se  $pos>0$ , por exemplo  $pos=2$

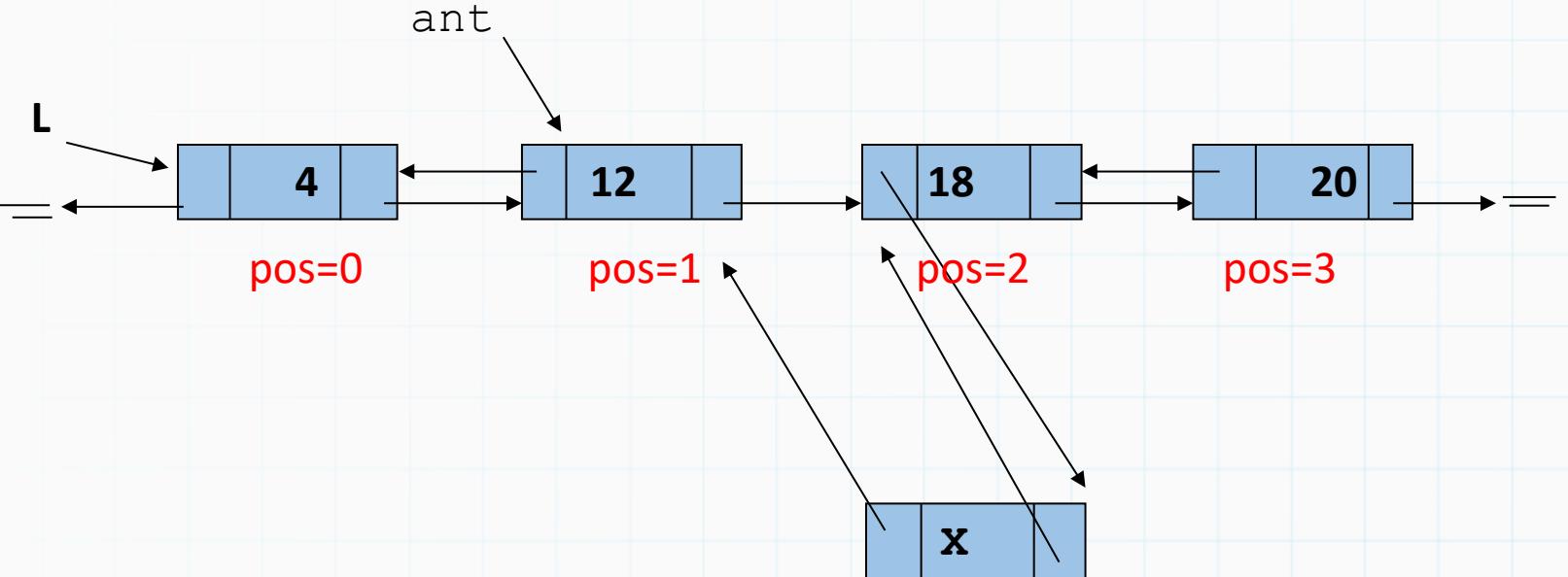


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant      = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se pos>0, por exemplo pos=2

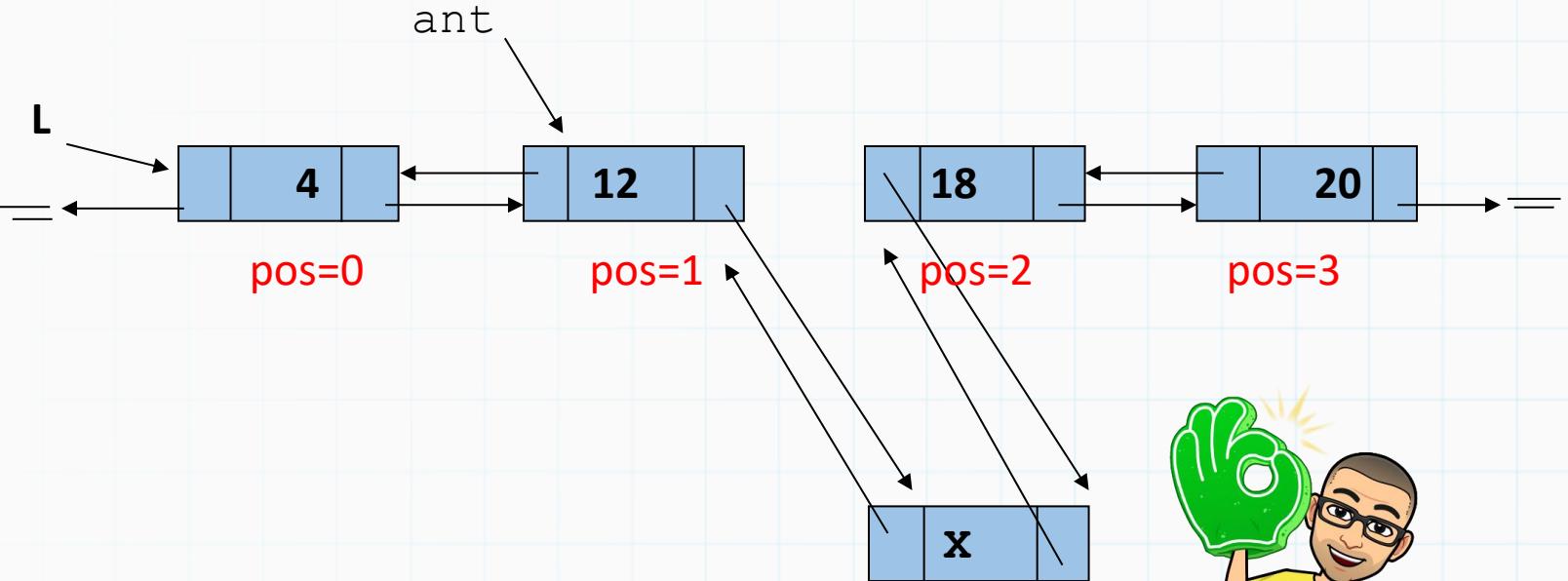


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se pos>0, por exemplo pos=2

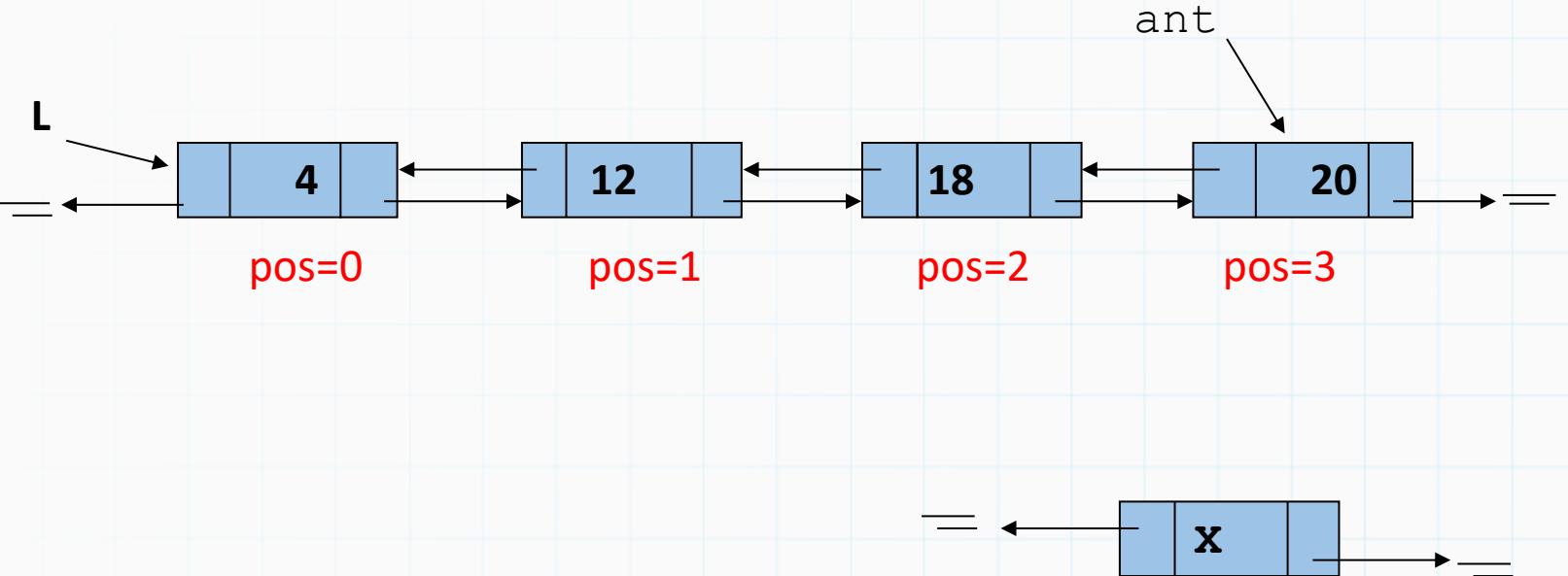


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se pos>0, por exemplo pos=4, no fim da lista



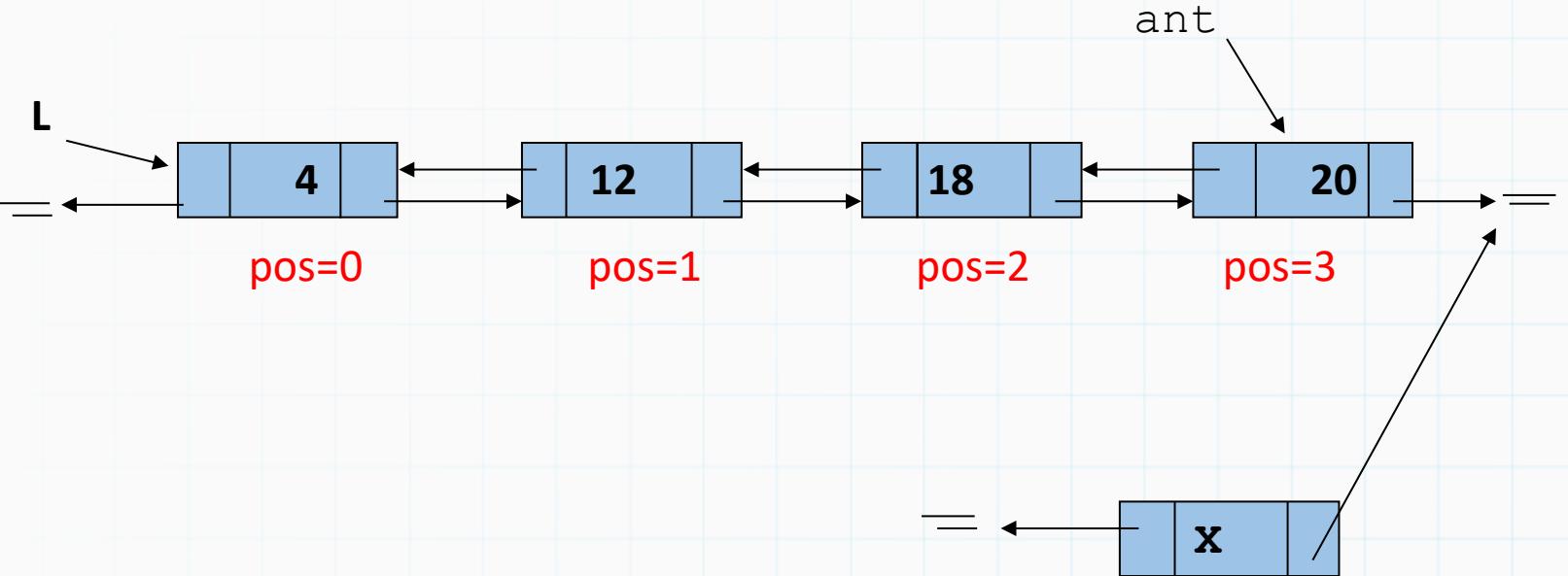
Vai funcionar ?

```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

Se pos>0, por exemplo pos=4, no fim da lista



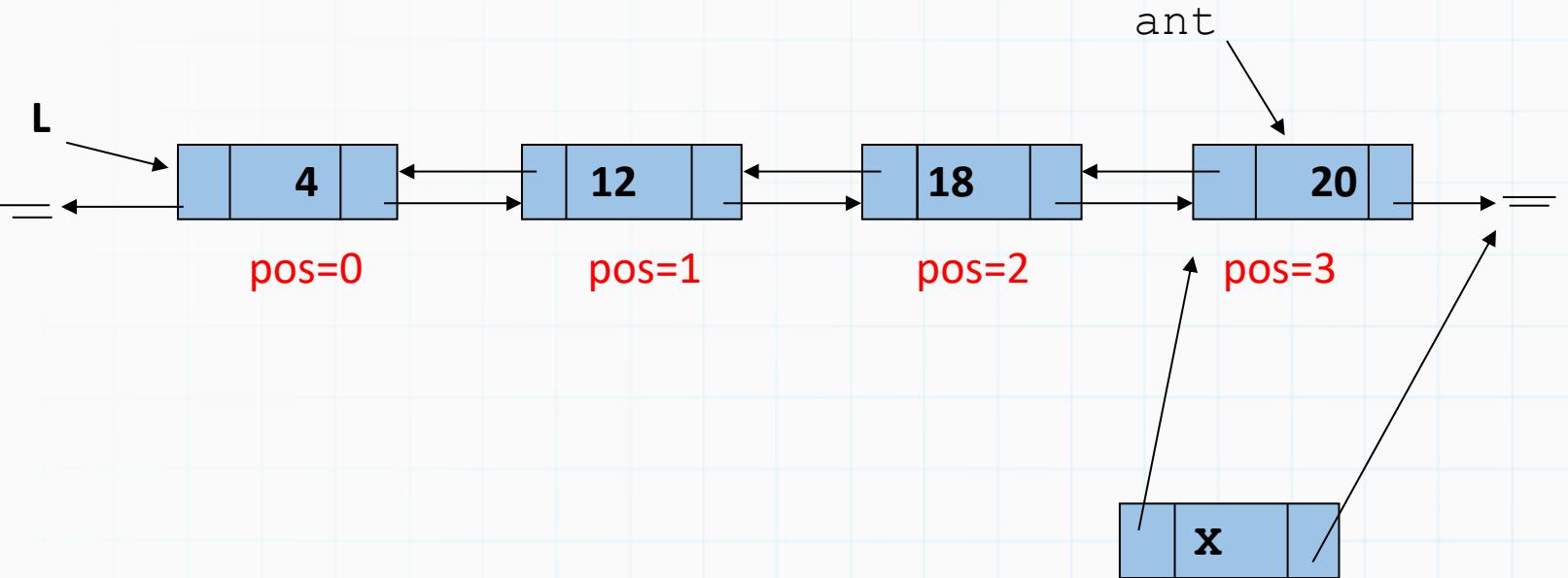
Vai funcionar ?

```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

Se  $pos>0$ , por exemplo  $pos=4$ , no fim da lista



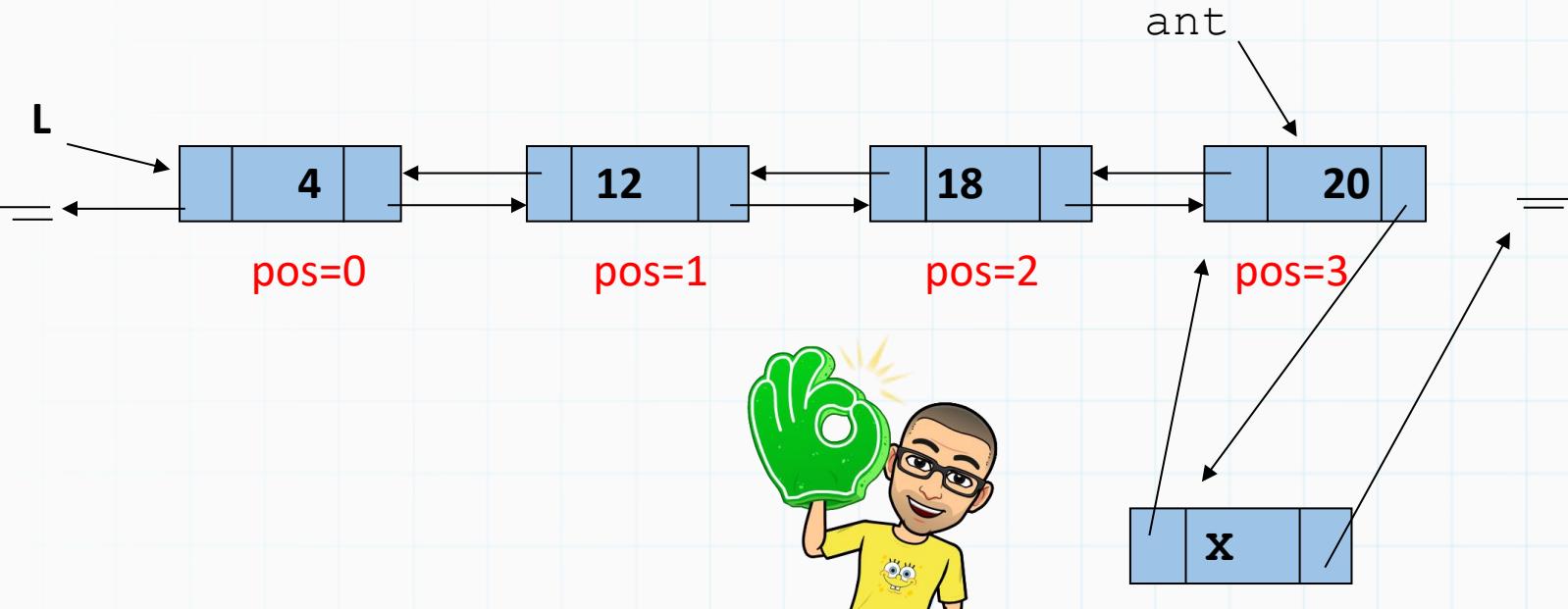
Vai funcionar ?

```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant      = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

Se  $pos>0$ , por exemplo  $pos=4$ , no fim da lista



Vai funcionar ?

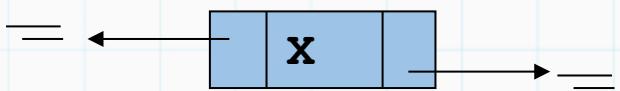
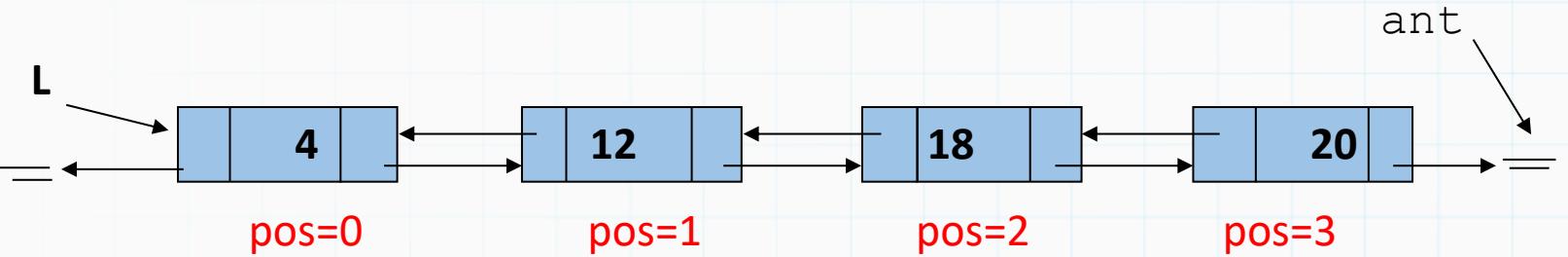
```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```



# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

Se  $pos>0$ , por exemplo  $pos=5$ , além do tamanho da lista



ERRO!

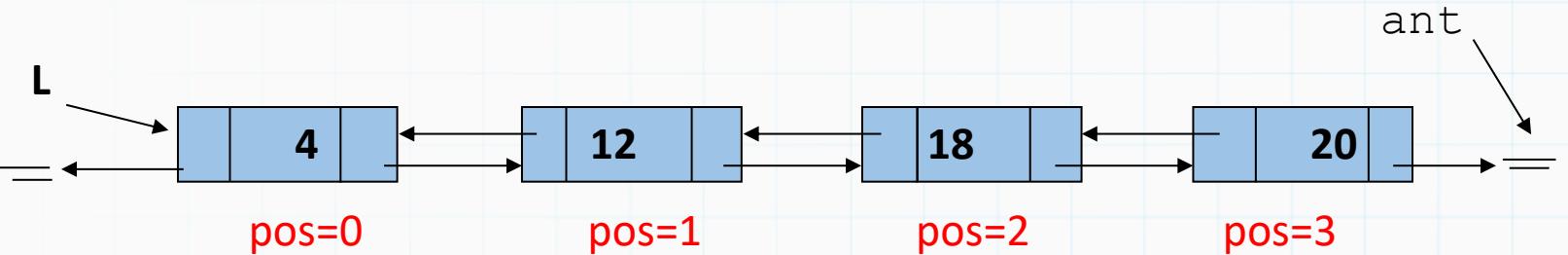


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

# Listas Duplamente Encadeadas

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

Se  $pos>0$ , por exemplo  $pos=5$ , além do tamanho da lista



```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
  
    // passou do fim  
    if (anterior == NULL)  
    {  
        printf("Posicao invalida\n");  
        return L;  
    }  
}  
  
lista *novo      = aloca_no();  
novo->info      = el;  
novo->prox      = anterior->prox;  
novo->ant       = anterior;  
  
if (anterior->prox != NULL)  
    anterior->prox->ant = novo;  
  
anterior->prox = novo;  
return L;
```

```
lista* insere_lista_pos(lista* L, int el, int pos)
{
    if(pos < 0)
    {
        printf("Posicao invalida\n");
        return L;
    }

    // insercao no inicio
    if ((pos == 0) || (L == NULL))
    {
        lista *no;
        no = aloca_no();
        no->info = el;
        no->prox = L;
        no->ant = NULL;

        if (L != NULL)
            L->ant = no;

        return no;
    }
}
```

```
// demais posicoes
else
{
    lista * anterior = L;

    // encontra anterior do elemento (se e
    for(int i=0; i<pos-1; i++)
    {
        anterior=anterior->prox;
    }

    // passou do fim
    if (anterior == NULL)
    {
        printf("Posicao invalida\n");
        return L;
    }

    lista *novo = aloca_no();
    novo->info = el;
    novo->prox = anterior->prox;
    novo->ant = anterior;

    if (anterior->prox != NULL)
        anterior->prox->ant = novo;

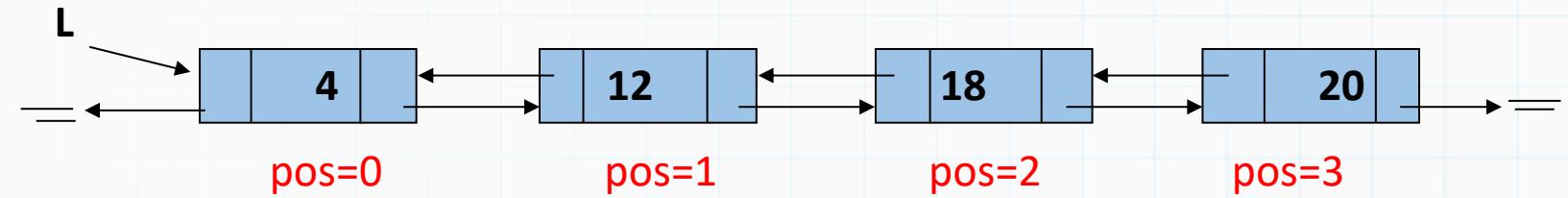
    anterior->prox = novo;
}

return L;
}
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



```
if (pos == 0)
{
    pt     = L;
    L     = L->prox;

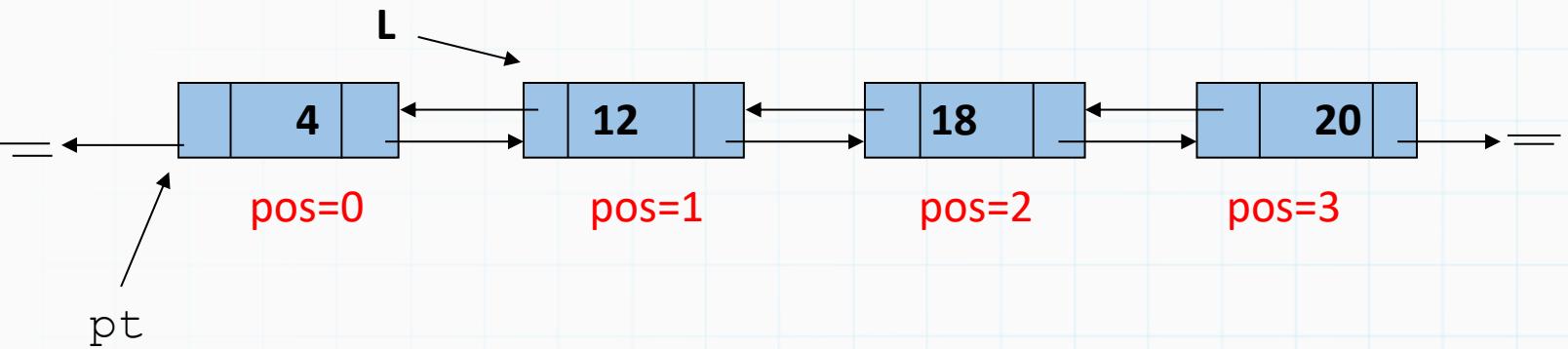
    if (L != NULL)
        L->ant = NULL;

    free(pt);
    return L;
}
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



```
if (pos == 0)
{
    pt      = L;
    L      = L->prox;

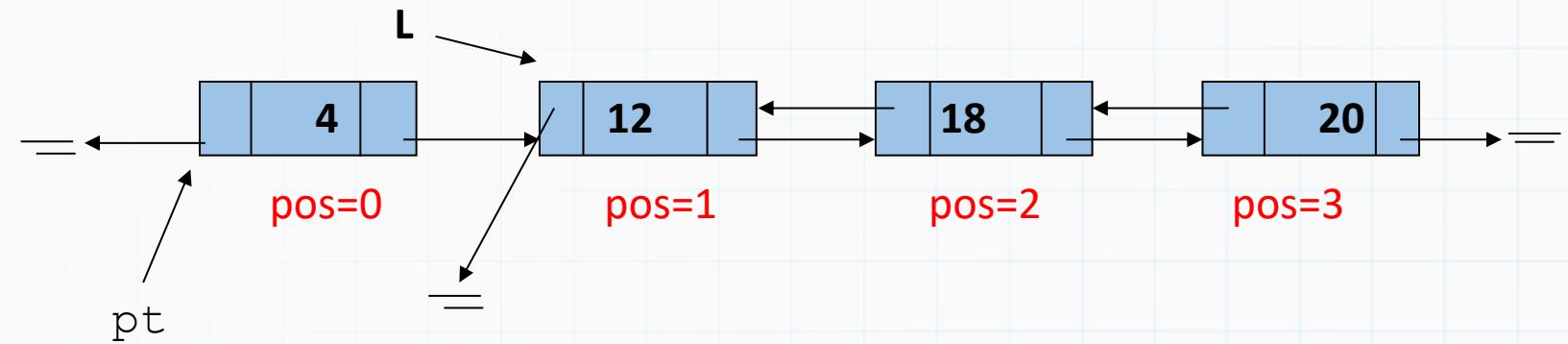
    if (L != NULL)
        L->ant = NULL;

    free(pt);
    return L;
}
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



```
if (pos == 0)
{
    pt      = L;
    L      = L->prox;

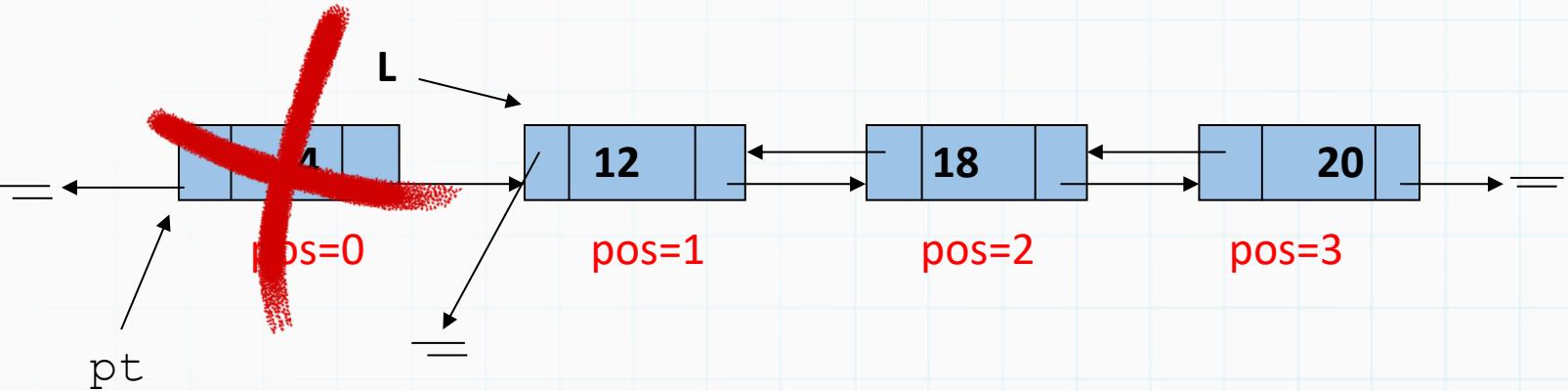
    if (L != NULL)
        L->ant = NULL;

    free(pt);
    return L;
}
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos=0$ , remove a cabeça da lista



```
if (pos == 0)
{
    pt      = L;
    L      = L->prox;

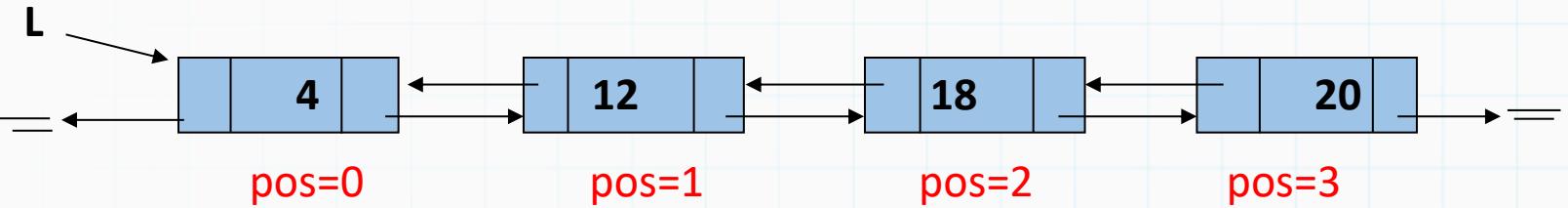
    if (L != NULL)
        L->ant = NULL;

    free(pt);
    return L;
}
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

pos>0, (encontra a posição) Ex: Queremos remover na pos = 2



Veja que agora que a lista é duplamente encadeada, não precisamos mais encontrar o anterior, apenas o elemento a ser removido

```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

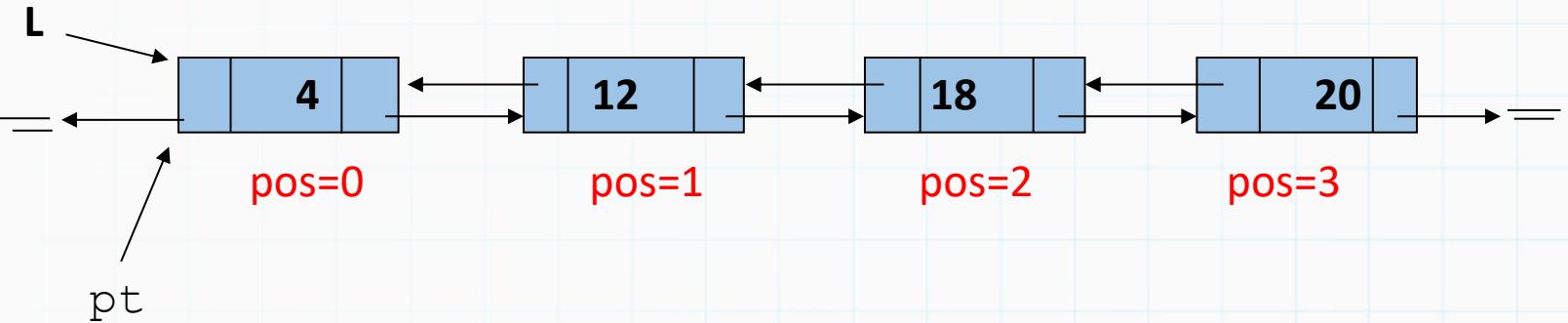
if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos>0$ , (encontra a posição) Ex: Queremos remover na  $pos = 2$



Veja que agora que a lista é duplamente encadeada, não precisamos mais encontrar o anterior, apenas o elemento a ser removido

```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

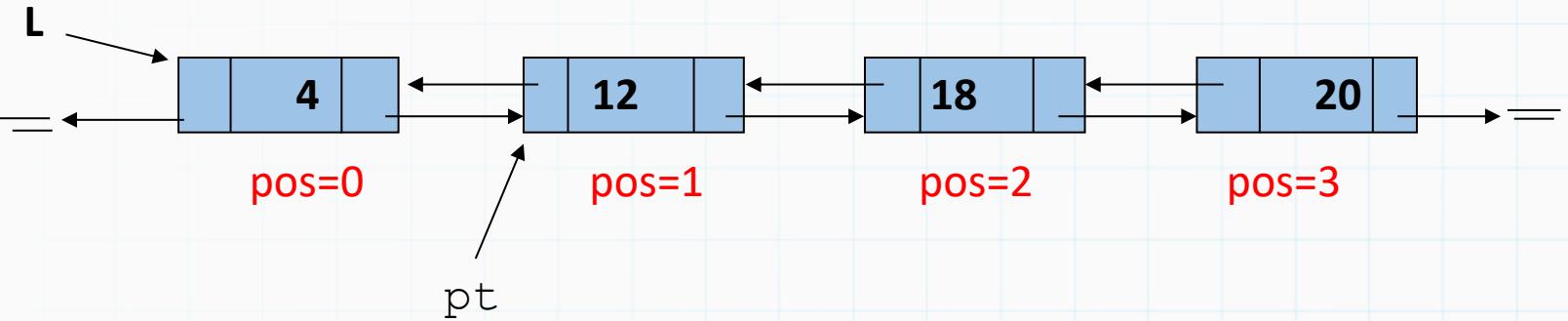
if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```

# Listas Duplamente Encadeadas

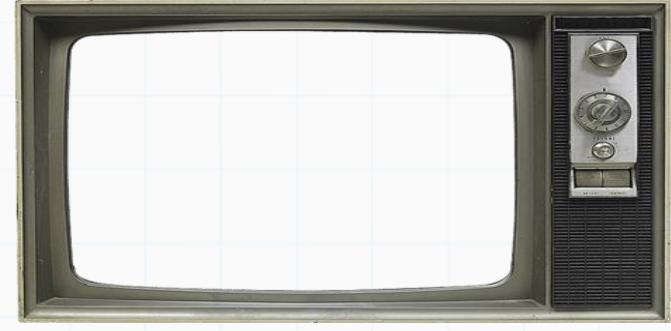
Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos > 0$ , (encontra a posição) Ex: Queremos remover na  $pos = 2$



$i=0$

Veja que agora que a lista é duplamente encadeada, não precisamos mais encontrar o anterior, apenas o elemento a ser removido



```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

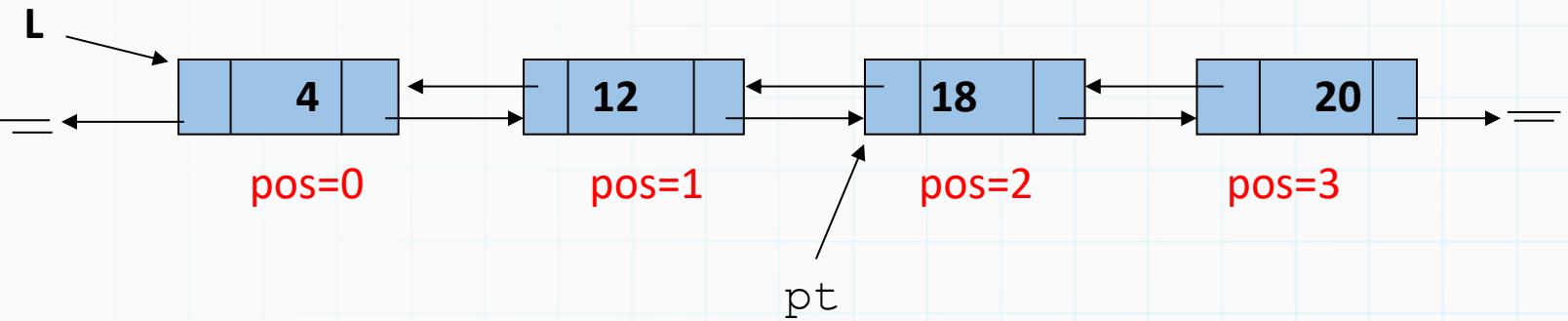
if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos > 0$ , (encontra a posição) Ex: Queremos remover na  $pos = 2$



Veja que agora que a lista é duplamente encadeada, não precisamos mais encontrar o anterior, apenas o elemento a ser removido

```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

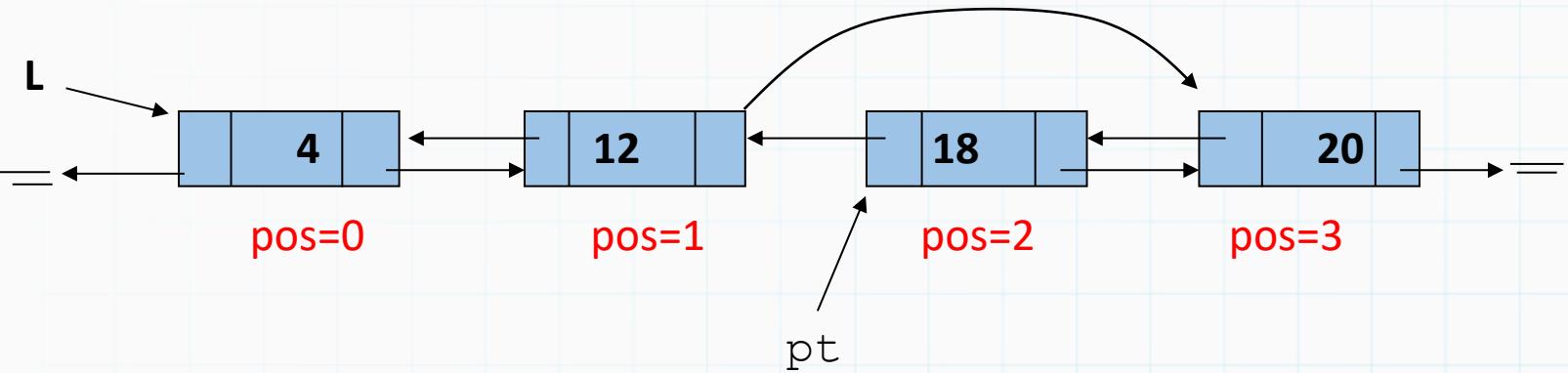
if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos>0$ , (encontra a posição) Ex: Queremos remover na  $pos = 2$



Veja que agora que a lista é duplamente encadeada, não precisamos mais encontrar o anterior, apenas o elemento a ser removido

```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

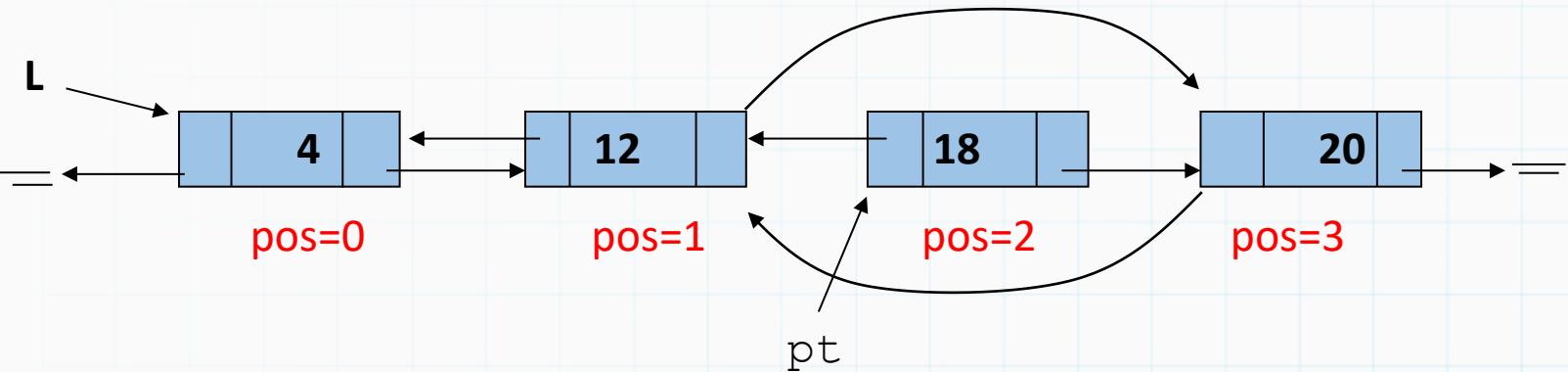
if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos > 0$ , (encontra a posição) Ex: Queremos remover na  $pos = 2$



Veja que agora que a lista é duplamente encadeada, não precisamos mais encontrar o anterior, apenas o elemento a ser removido

```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

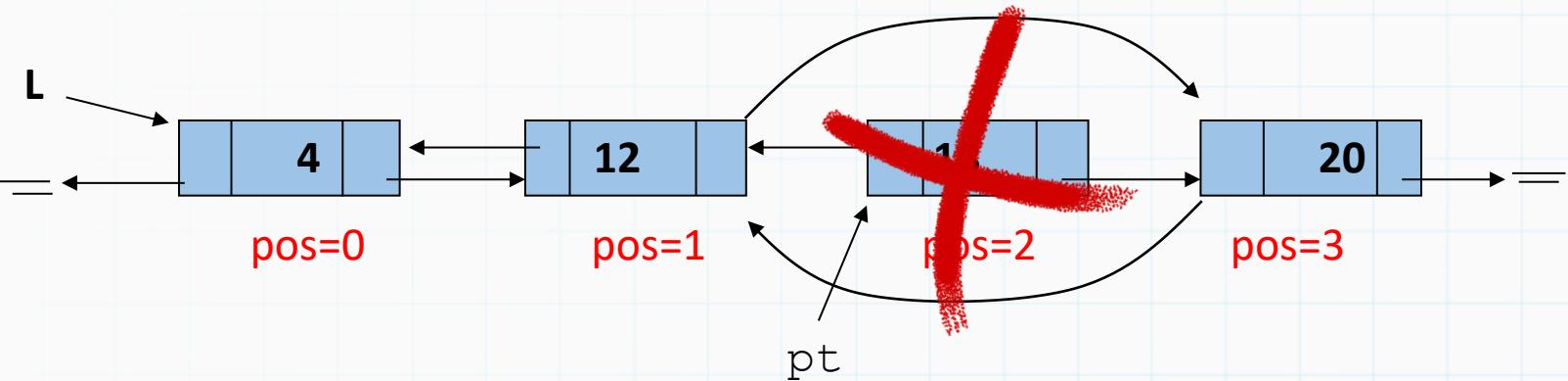
if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos > 0$ , (encontra a posição) Ex: Queremos remover na  $pos = 2$



Veja que agora que a lista é duplamente encadeada, não precisamos mais encontrar o anterior, apenas o elemento a ser removido

```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

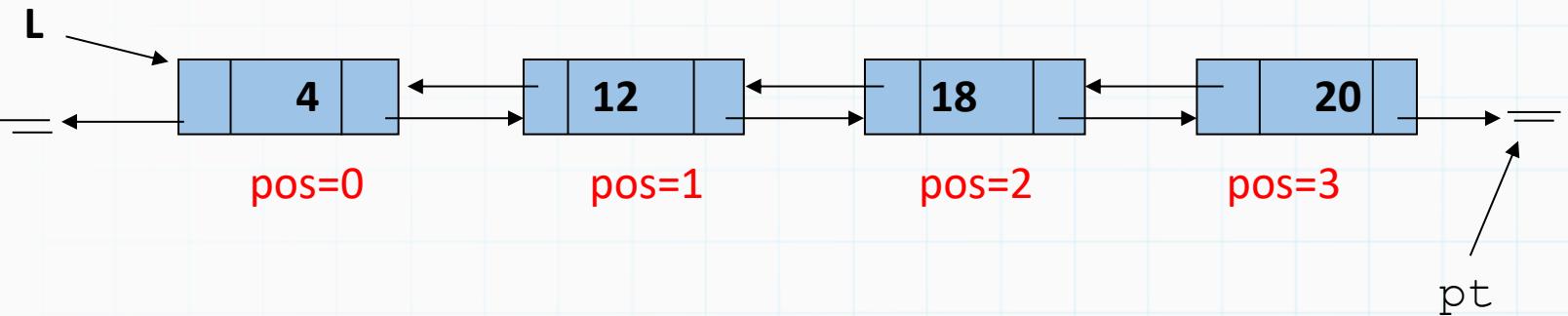
if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos > n-1$ , Ex: Queremos remover na  $pos = 4$



```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

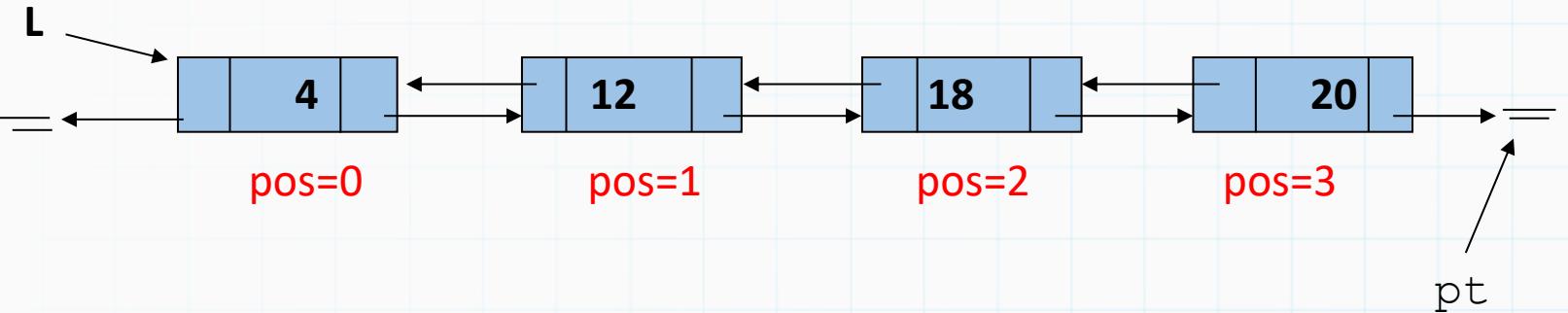
if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```

# Listas Duplamente Encadeadas

Remover: A remoção de um elemento numa posição específica  $pos=\{0, 1, 2, \dots, n-1\}$

$pos > n-1$ , Ex: Queremos remover na  $pos = 4$



```
pt = L;
for(int i=0; i<pos; i++)
{
    pt      = pt->prox;

    // passou do fim
    if (pt == NULL)
    {
        printf("Posicao invalida\n");
        return L;
    }
}

if (pt->ant != NULL)
    pt->ant->prox = pt->prox;

if (pt->prox != NULL)
    pt->prox->ant = pt->ant;

free(pt);
return L;
```



```

lista* remove_lista_pos_D(lista* L, int pos)
{
    lista * pt;

    if ((pos < 0) || (L == NULL))
    {
        printf("Posicao invalida/Lista vazia\n");
        return L;
    }

    // remocao no inicio
    if (pos == 0)
    {
        pt      = L;
        L      = L->prox;

        if (L != NULL)
            L->ant = NULL;

        free(pt);
        return L;
    }
}

```

```

    // demais posicoes
    else
    {
        pt = L;
        // encontra posicao do elemento
        for(int i=0; i<pos; i++)
        {
            pt      = pt->prox;

            // passou do fim
            if (pt == NULL)
            {
                printf("Posicao invalida\n");
                return L;
            }
        }

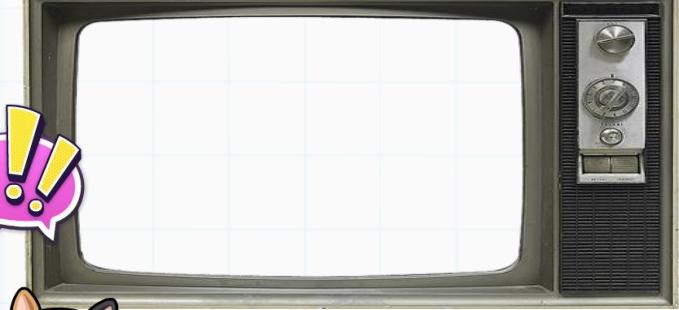
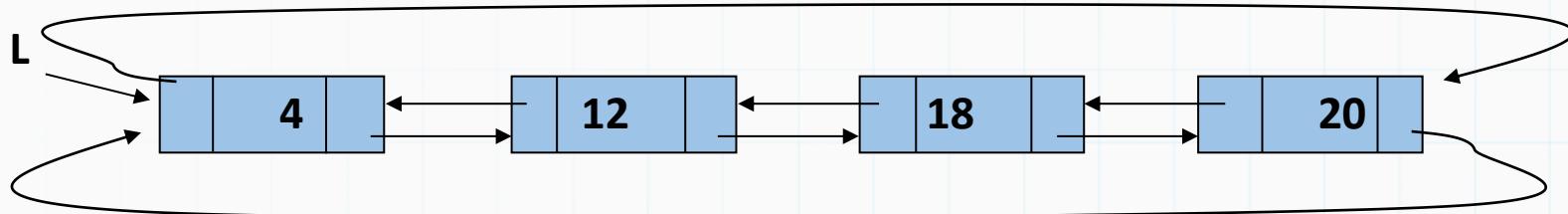
        if (pt->ant != NULL)
            pt->ant->prox = pt->prox;
        if (pt->prox != NULL)
            pt->prox->ant = pt->ant;

        free(pt);
        return L;
    }
}

```

# Listas Duplamente Encadeadas Circular

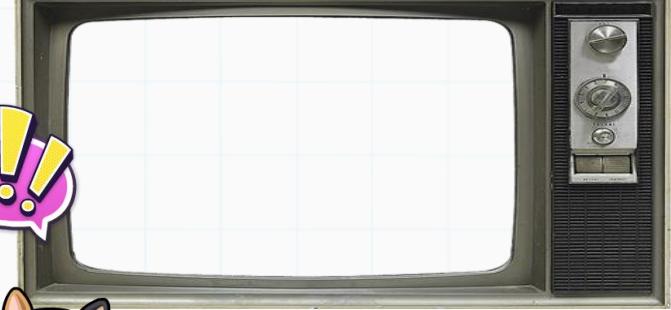
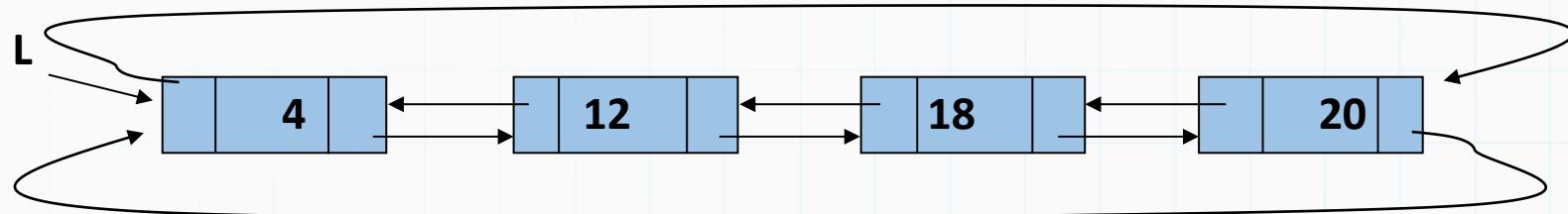
Vamos juntar tudo, duplamente encadeada e circular



**esta não é nem minha forma final**

# Listas Duplamente Encadeadas Circular

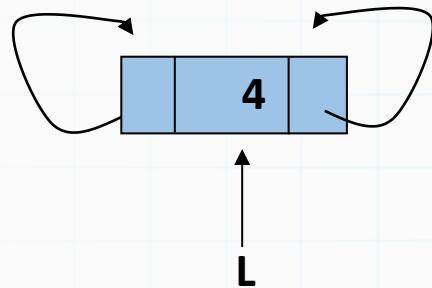
Vamos juntar tudo, duplamente encadeada e circular



Exemplo:

apenas um no

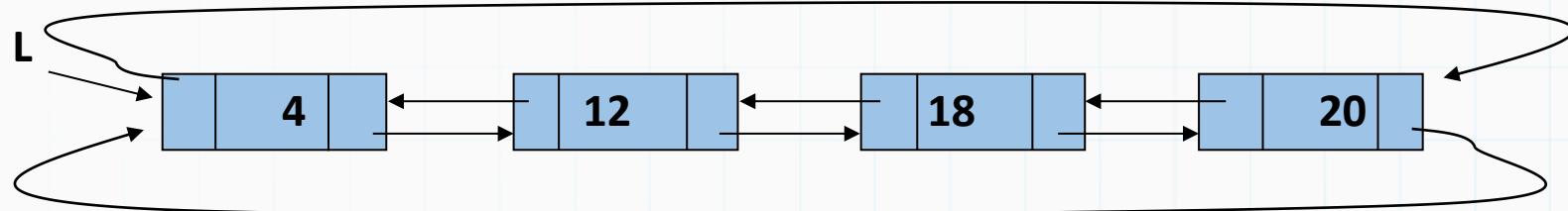
Vazia



esta não é nem minha forma final

# Listas Duplamente Encadeadas Circular

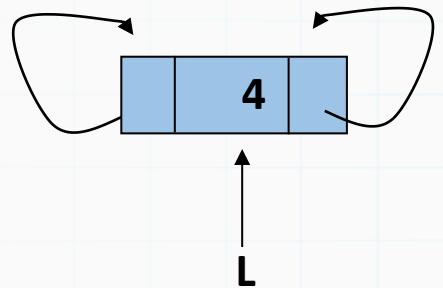
Vamos juntar tudo, duplamente encadeada e circular



Exemplo:

apenas um no

Vazia



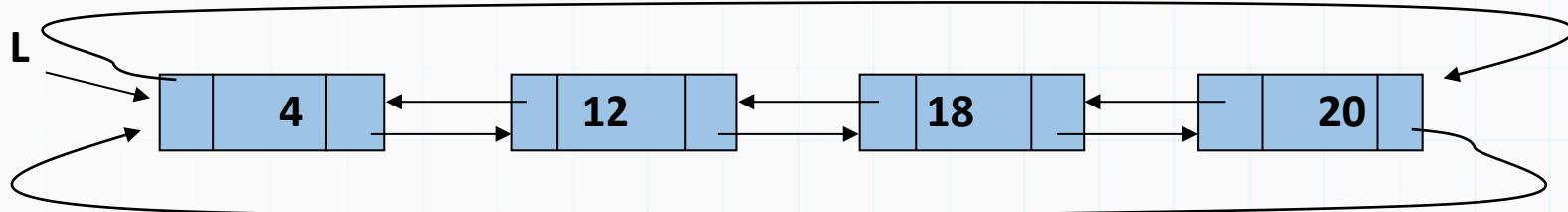
```
struct NO {  
    int info;  
    struct NO *prox, *ant;  
}  
typedef struct NO lista;  
  
...  
lista *L;  
L = (lista*) malloc(sizeof(lista));  
L->prox = L;  
L->ant = L;
```

Estrutura é a mesma da lista circular



# Listas Duplamente Encadeadas Circular

Vamos juntar tudo, duplamente encadeada e circular



Percorrer: Igual a circular

```
void imprime_lista_DC(lista* L)
{
    lista* no = L;

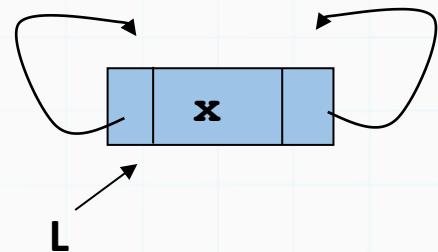
    printf("L = ");
    while (no != NULL)
    {
        printf("%d, ", no->info);
        no = no->prox;
    }
    printf("\n");
}
```



# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ...,n}

para L = vazia



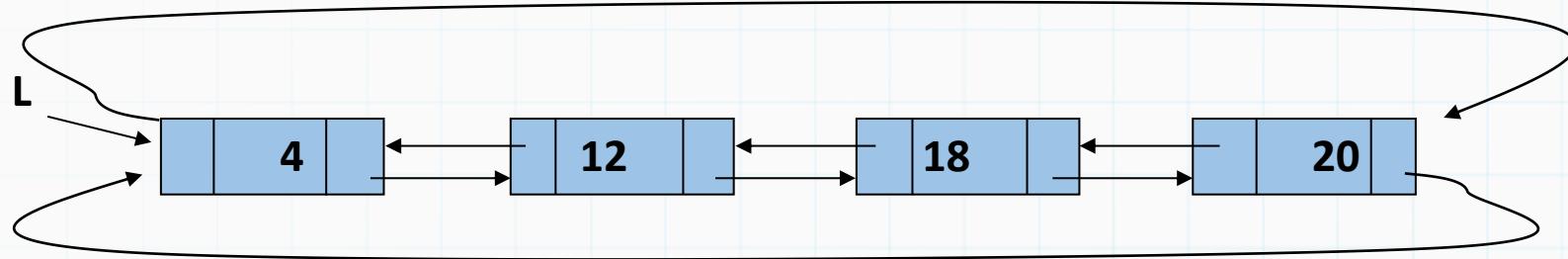
```
if (L == NULL)
{
    lista *no;
    no      = aloca_no();
    no->info = el;
    no->prox = no;
    no->ant  = no;
    return no;
}
```

# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}



para pos=0



```
if (pos == 0)
{
    lista *no;
    no      = aloca_no();
    no->info = el;

    lista * ultimo = L->ant;
    no->prox      = L;
    no->ant       = ultimo;
    L->ant        = no;
    ultimo->prox  = no;

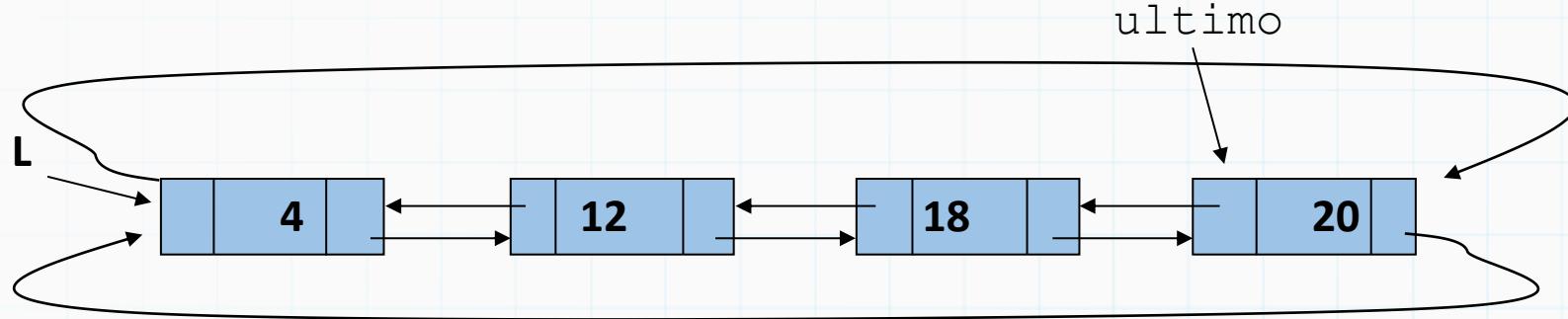
    return no;
}
```

# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}



para pos=0



```
if (pos == 0)
{
    lista *no;
    no      = aloca_no();
    no->info = el;

    lista * ultimo = L->ant;
    no->prox      = L;
    no->ant       = ultimo;
    L->ant        = no;
    ultimo->prox  = no;

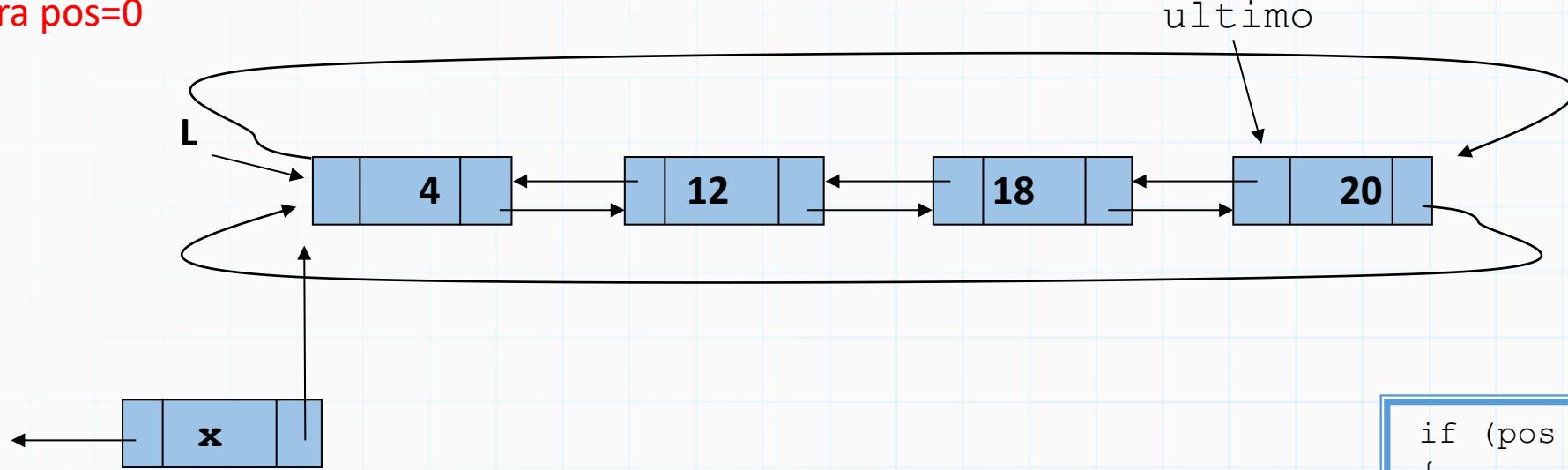
    return no;
}
```

# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}



para pos=0



```
if (pos == 0)
{
    lista *no;
    no      = aloca_no();
    no->info = el;

    lista * ultimo = L->ant;
    no->prox      = L;
    no->ant       = ultimo;
    L->ant        = no;
    ultimo->prox  = no;

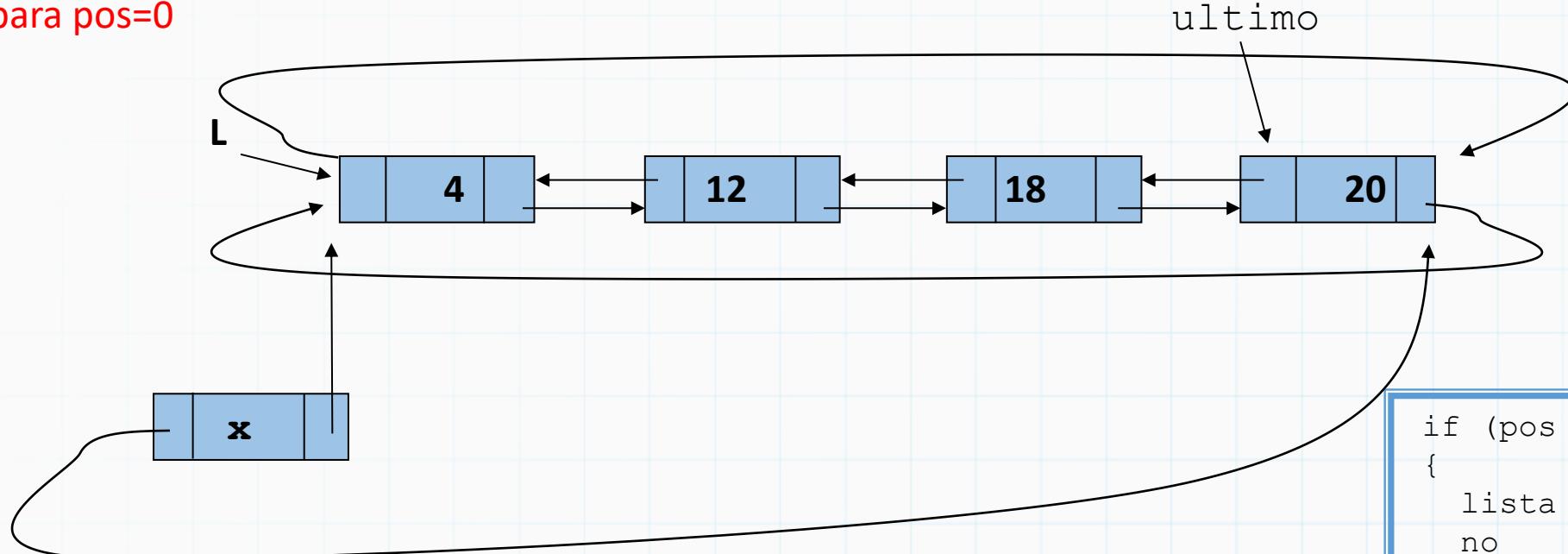
    return no;
}
```

# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}



para pos=0



```
if (pos == 0)
{
    lista *no;
    no          = aloca_no();
    no->info  = el;

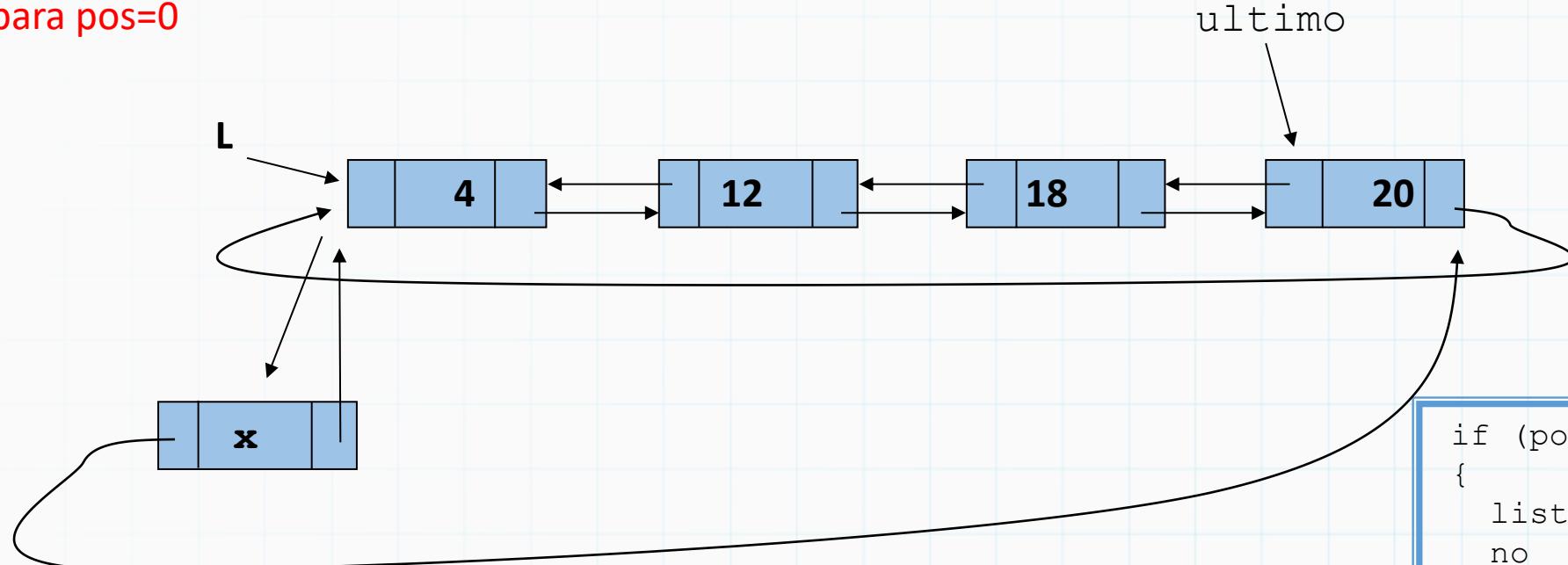
    lista * ultimo = L->ant;
    no->prox      = L;
    no->ant      = ultimo;
    L->ant        = no;
    ultimo->prox  = no;

    return no;
}
```

# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

para pos=0



```
if (pos == 0)
{
    lista *no;
    no      = aloca_no();
    no->info = el;

    lista * ultimo = L->ant;
    no->prox      = L;
    no->ant       = ultimo;
L->ant      = no;
    ultimo->prox  = no;

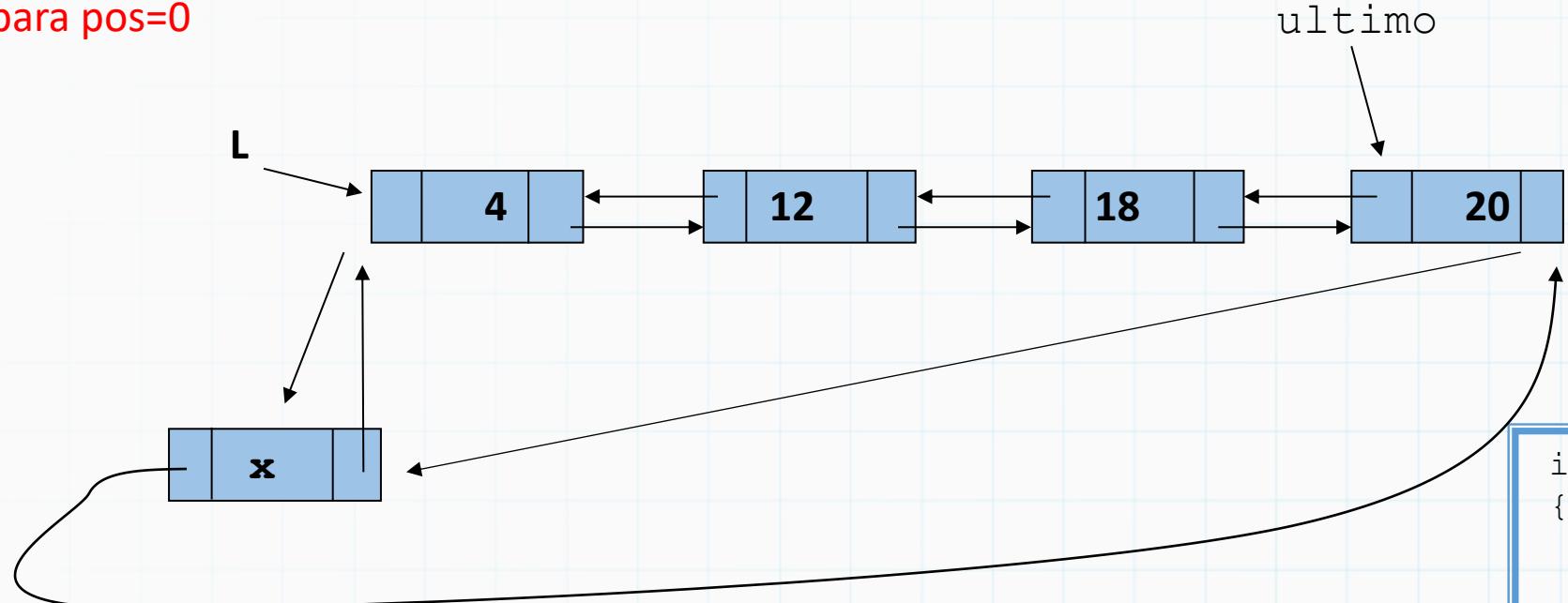
    return no;
}
```



# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

para pos=0



```
if (pos == 0)
{
    lista *no;
    no      = aloca_no();
    no->info = el;

    lista * ultimo = L->ant;
    no->prox      = L;
    no->ant       = ultimo;
    L->ant        = no;
    ultimo->prox  = no;

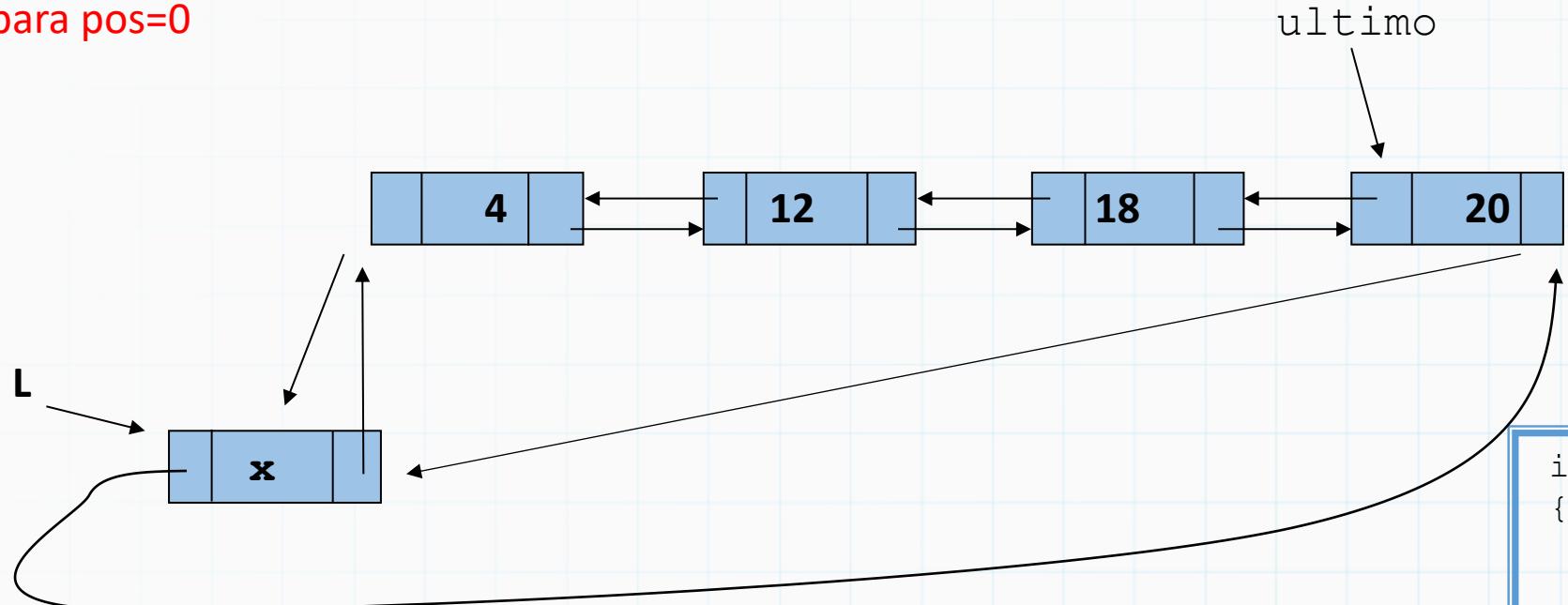
    return no;
}
```



# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

para pos=0



```
if (pos == 0)
{
    lista *no;
    no      = aloca_no();
    no->info = el;

    lista * ultimo = L->ant;
    no->prox      = L;
    no->ant       = ultimo;
    L->ant        = no;
    ultimo->prox  = no;

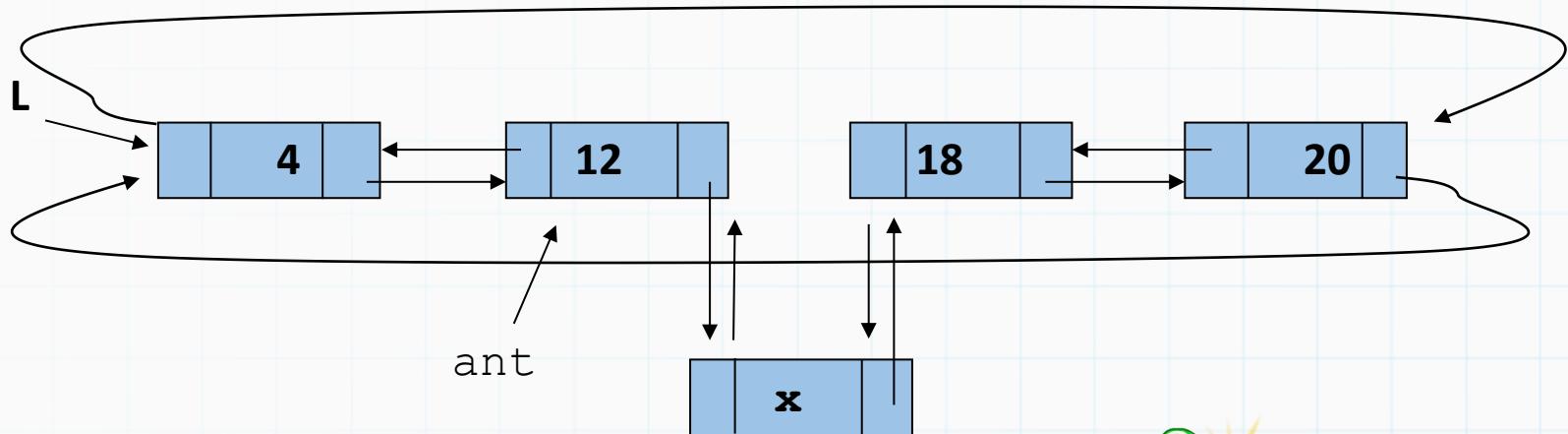
    return no;
}
```



# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica pos={0, 1, 2, ..., n}

para  $n \geq pos > 0 \rightarrow$  igual a duplamente encadeado

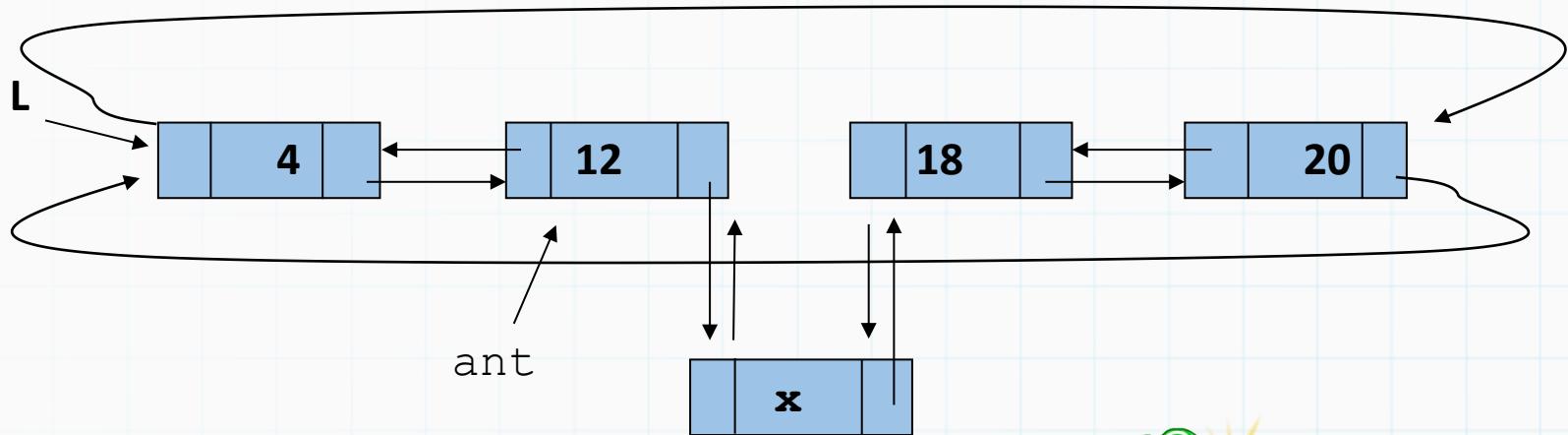


```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
  
    // passou do fim  
    if (anterior == L)  
    {  
        printf("Posicao invalida\n");  
        return L;  
    }  
}  
  
lista *novo = aloca_no();  
novo->info = el;  
novo->prox = anterior->prox;  
novo->ant = anterior;  
anterior->prox->ant = novo;  
anterior->prox = novo;  
  
return L;
```

# Listas Duplamente Encadeadas Circular

Inserir: A inserção de um elemento X numa posição específica  $pos=\{0, 1, 2, \dots, n\}$

para  $n \geq pos > 0 \rightarrow$  igual a duplamente encadeado



- Remoção segue a mesma ideia da Inserção...



```
lista * anterior = L;  
  
for(int i=0; i<pos-1; i++)  
{  
    anterior=anterior->prox;  
  
    // passou do fim  
    if (anterior == L)  
    {  
        printf("Posicao invalida\n");  
        return L;  
    }  
}  
  
lista *novo = aloca_no();  
novo->info = el;  
novo->prox = anterior->prox;  
novo->ant = anterior;  
anterior->prox->ant = novo;  
anterior->prox = novo;  
  
return L;
```

```

lista* insere_lista_pos_DC(lista* L, int el, int pos)
{
    if(pos < 0)
    {
        printf("Posicao invalida\n");
        return L;
    }
    // insercao no inicio
    if (L == NULL)
    {
        lista *no;
        no      = aloca_no();
        no->info = el;
        no->prox = no;
        no->ant = no;
        return no;
    }
    // posicao inicial (cabeca)
    else if (pos == 0)
    {
        lista *no;
        no      = aloca_no();
        no->info = el;

        lista * ultimo = L->ant;
        no->prox      = L;
        no->ant       = ultimo;
        L->ant        = no;
        ultimo->prox = no;

        return no;
    }
}

```

```

// demais posicoes
else
{
    lista * anterior = L;

    // encontra anterior do elemento (se existir)
    for(int i=0; i<pos-1; i++)
    {
        anterior=anterior->prox;

        // passou do fim
        if (anterior == L)
        {
            printf("Posicao invalida\n");
            return L;
        }
    }

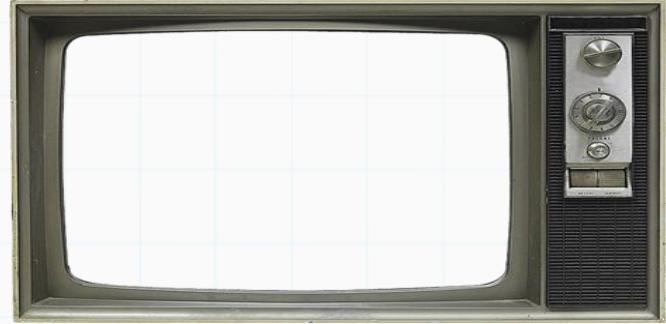
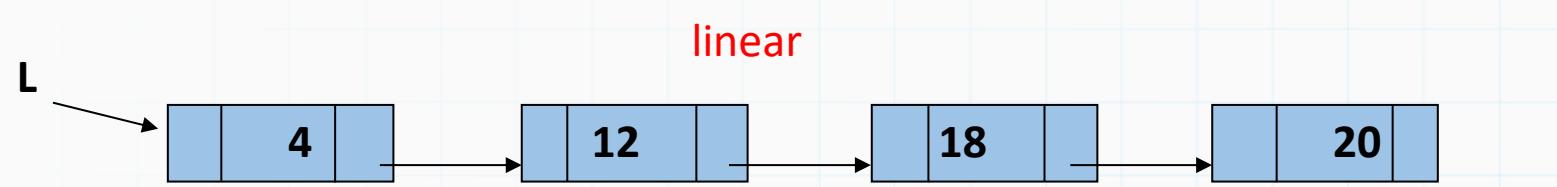
    lista *novo      = aloca_no();
    novo->info     = el;
    novo->prox     = anterior->prox;
    novo->ant      = anterior;
    anterior->prox->ant = novo;
    anterior->prox = novo;
}

return L;
}

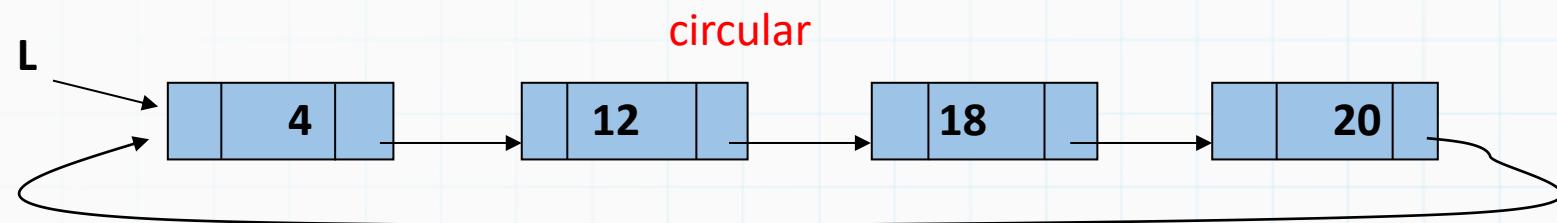
```

# Comparação entre listas

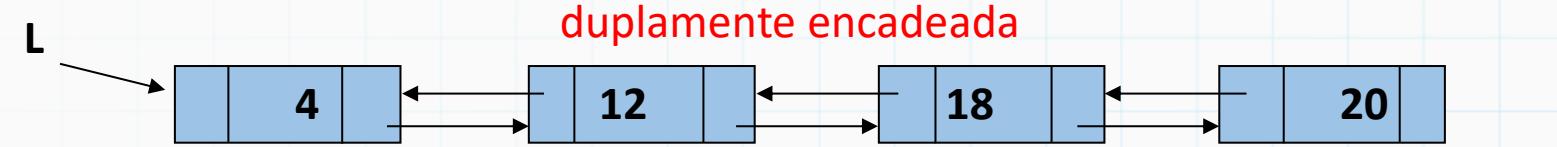
Espaço: lista de tamanho  $n$



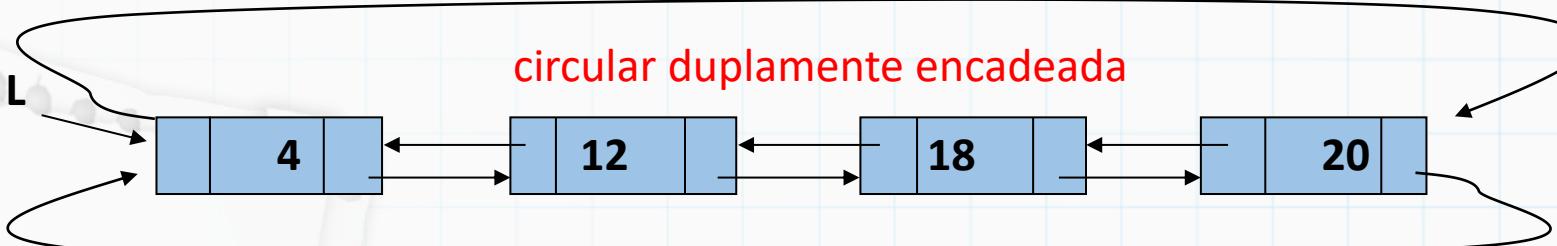
$n$  ponteiros



$n$  ponteiros ( $n+1$  para ser exato)



$2n$  ponteiros

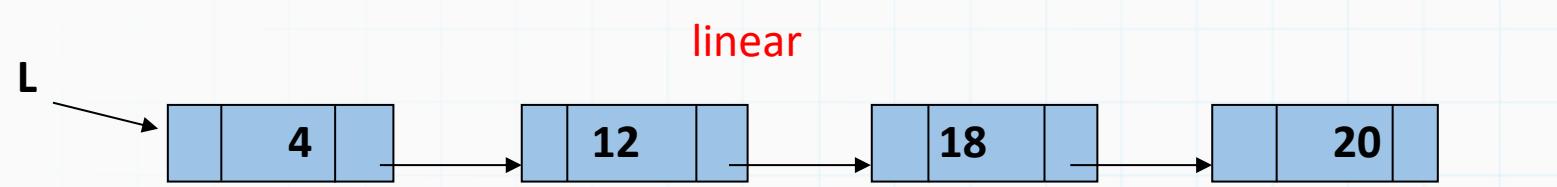


$2n$  ponteiros

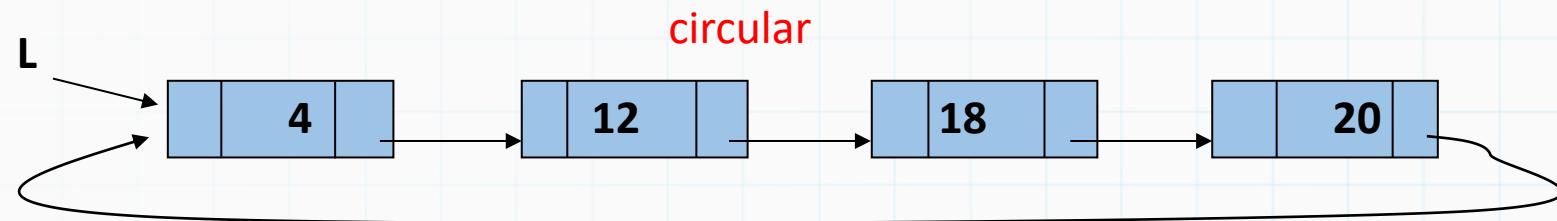


# Comparação entre listas

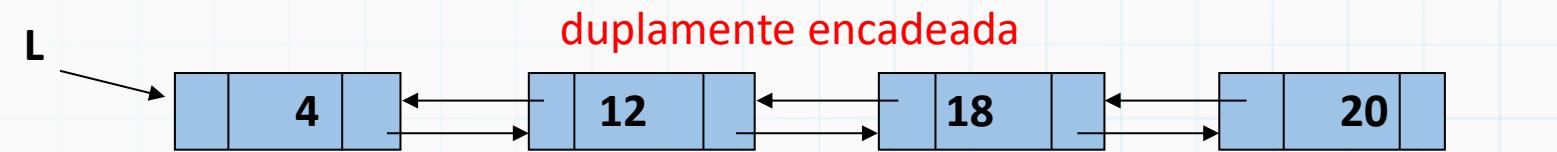
Inserção: na posição 0



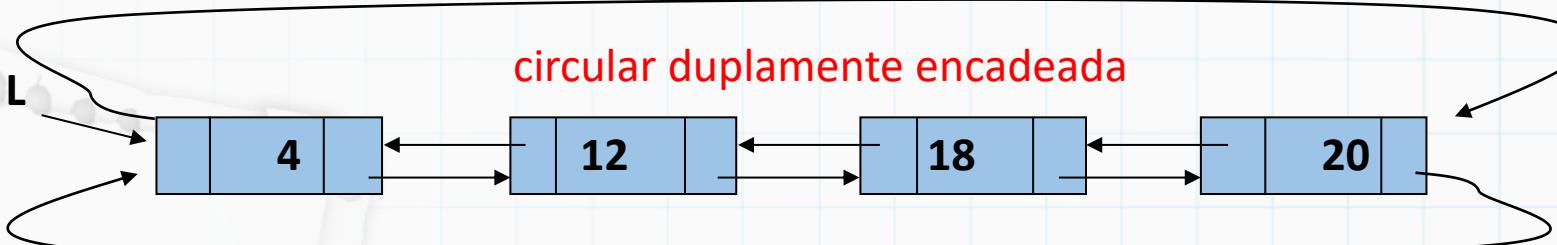
tempo constante



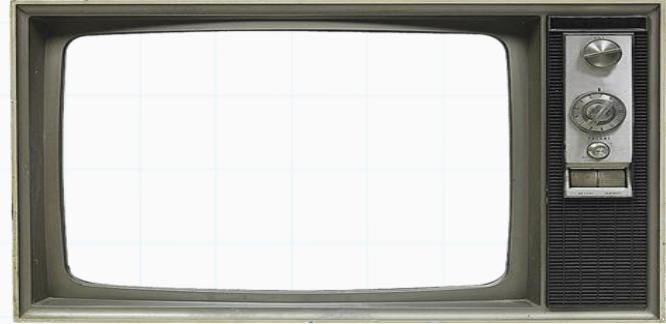
percorre n elementos (por causa do ponteiro final)



tempo constante

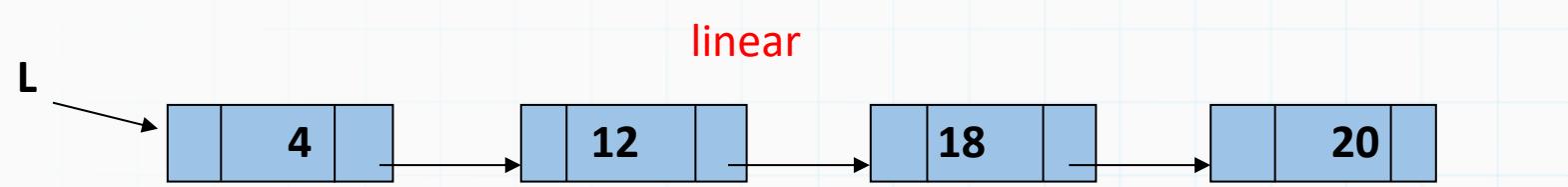


tempo constante

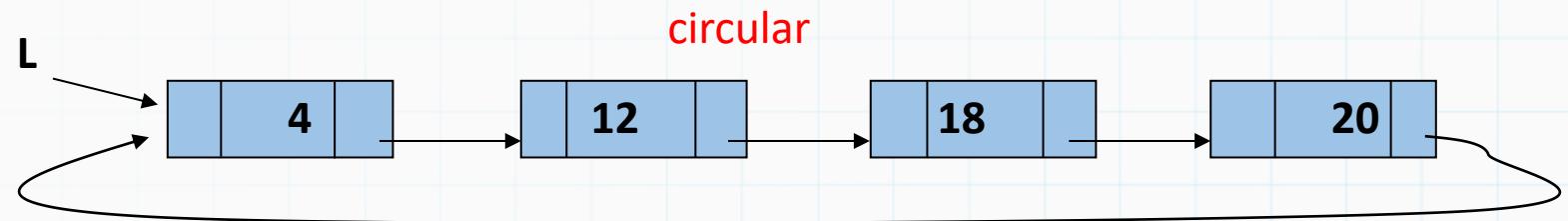


# Comparação entre listas

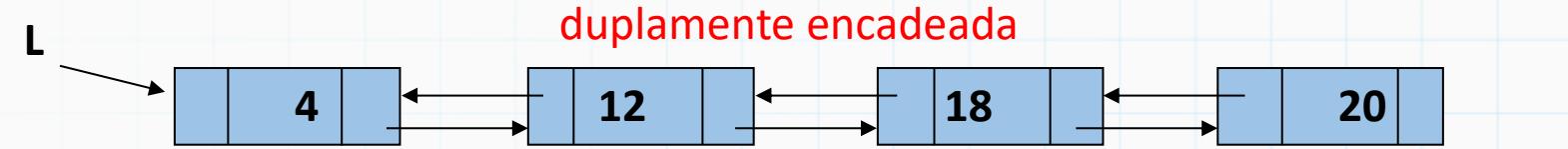
Inserção: na posição  $> 0$



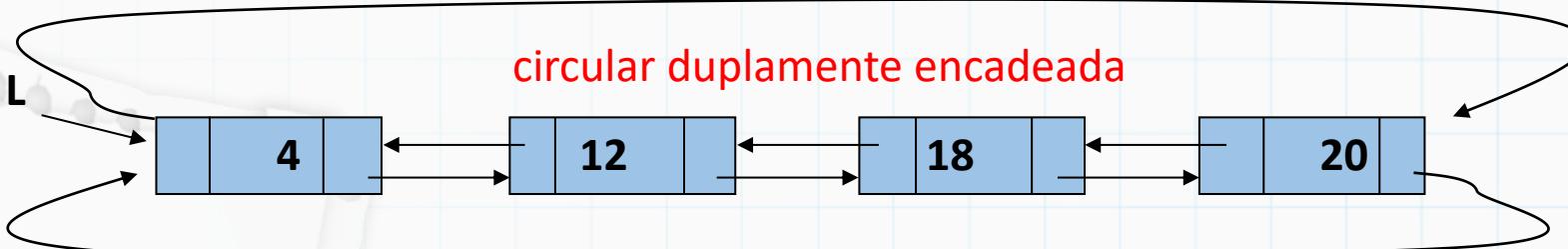
pioor caso percorre  $n$  elementos



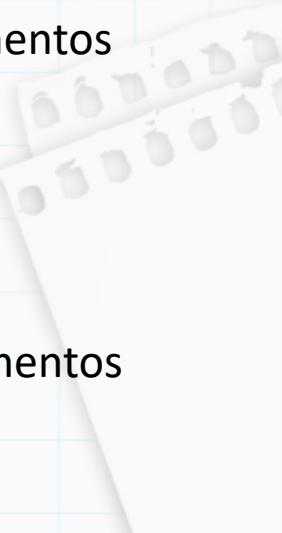
pioor caso percorre  $n$  elementos



pioor caso percorre  $n$  elementos

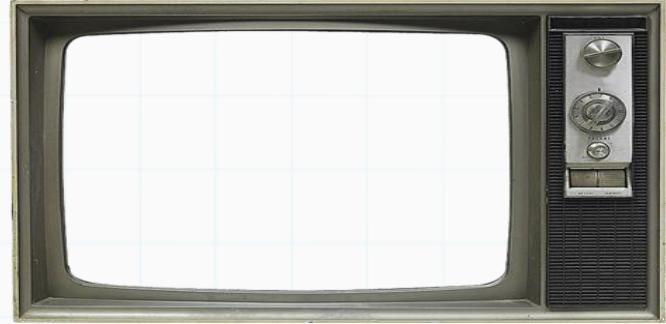
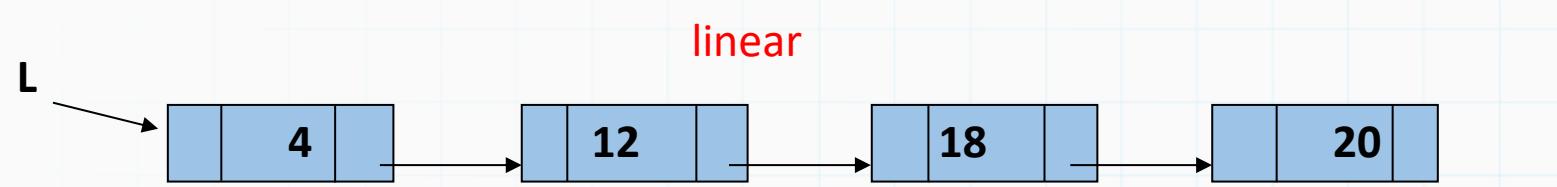


pioor caso percorre  $n$  elementos

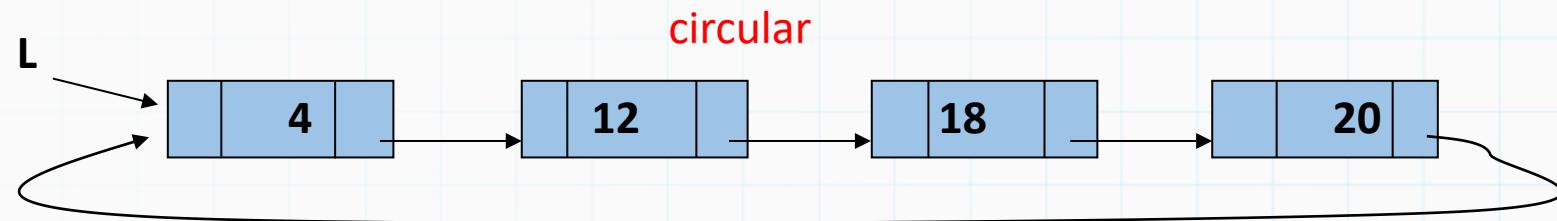


# Comparação entre listas

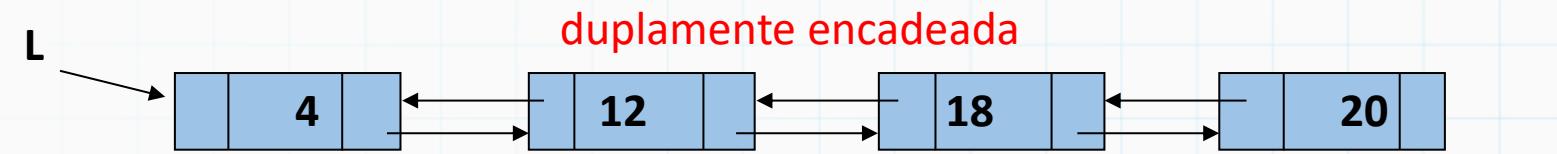
Remoção: na posição = 0



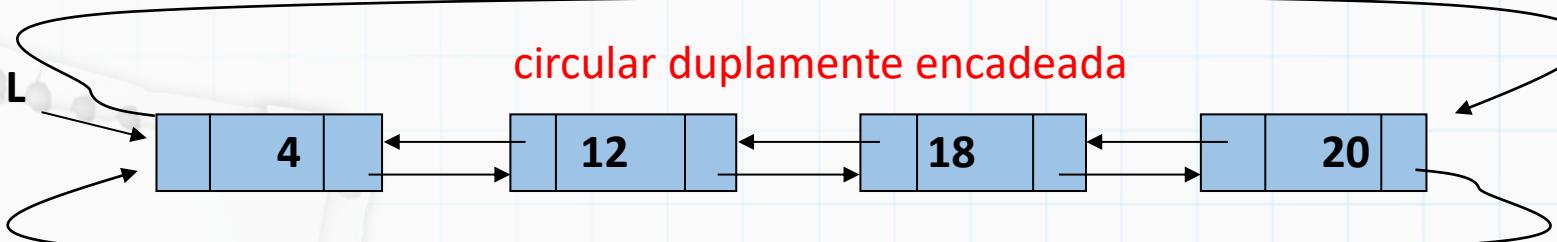
tempo constante



percorre n elementos (por causa do link final)



tempo constante

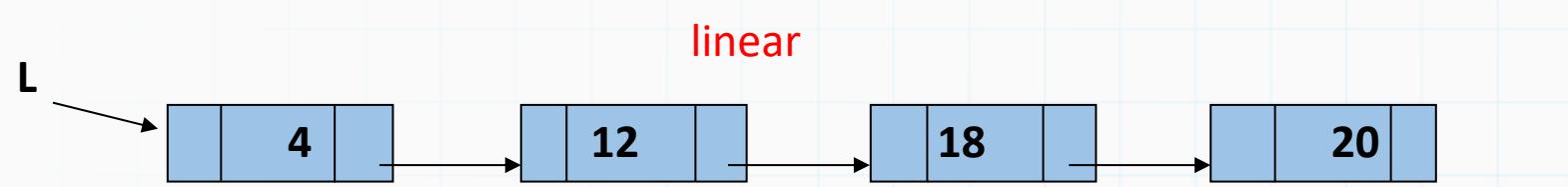


tempo constante

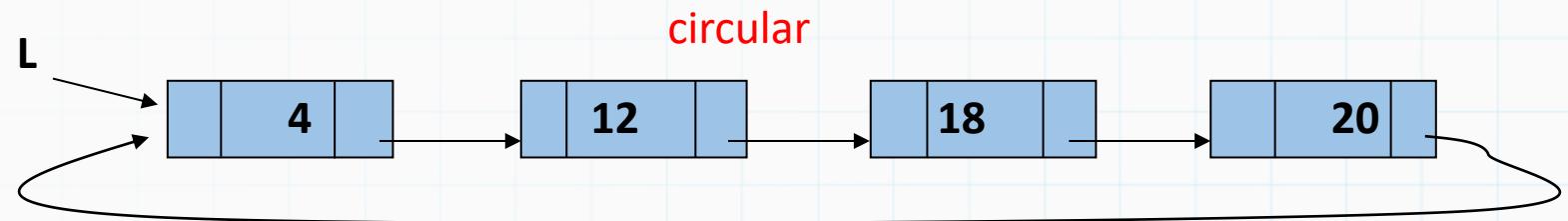


# Comparação entre listas

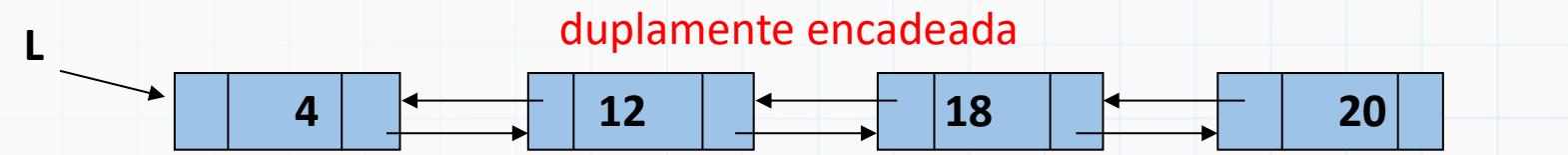
Remoção: na posição  $> 0$



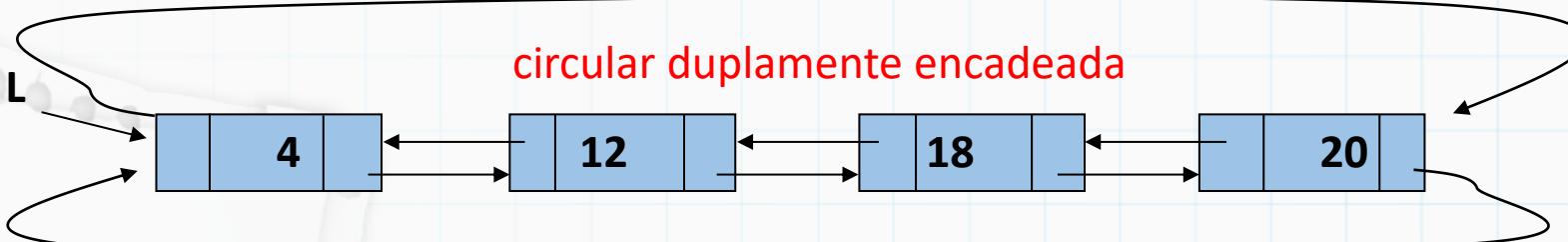
pioor caso percorre  $n$  elementos



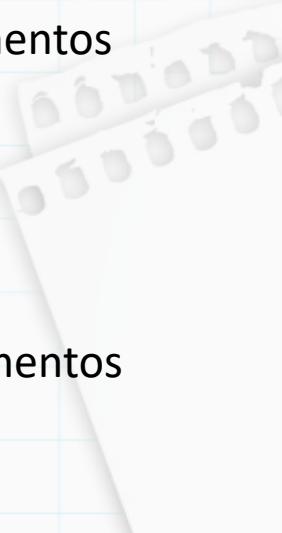
pioor caso percorre  $n$  elementos



pioor caso percorre  $n$  elementos

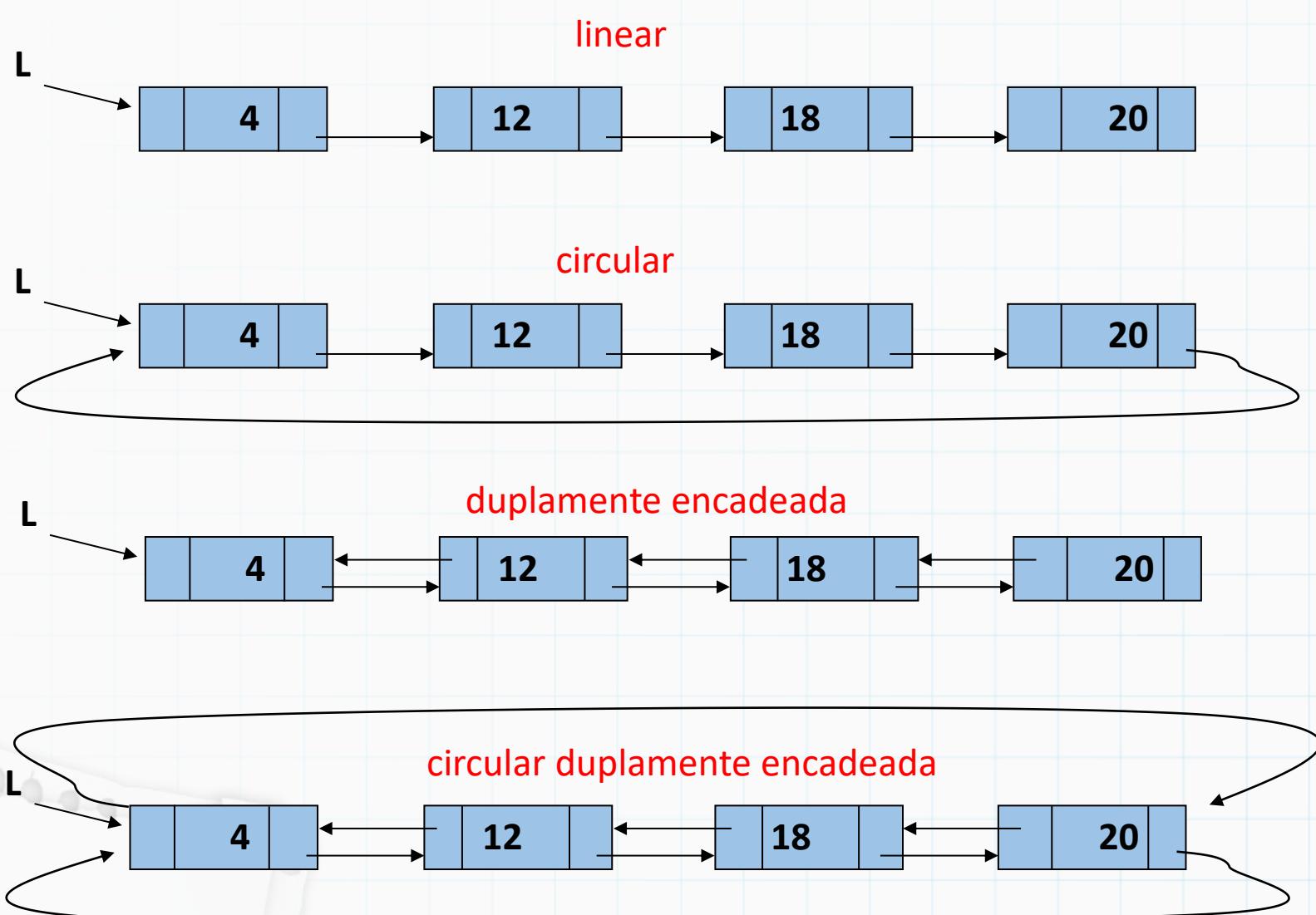


pioor caso percorre  $n$  elementos



# Comparação entre listas

Concluindo:



limitada, porém usa pouca memória

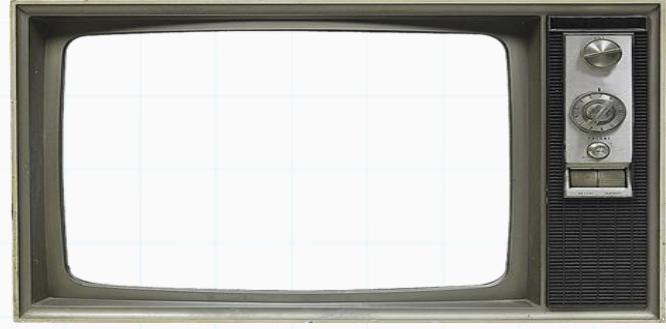
meio termo

meio termo

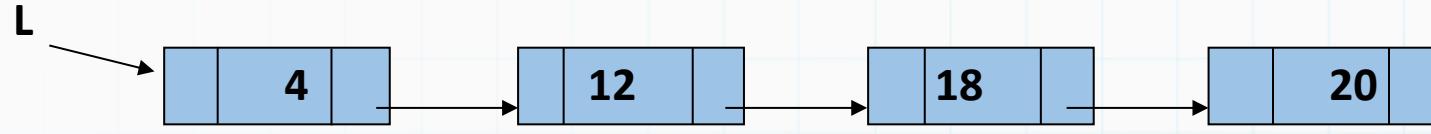
Mais eficiente/poderosa porém  
precisa de mais memória



# listas x vetores



Alocação:



- Tamanho variável, aumenta e diminui a vontade 😊

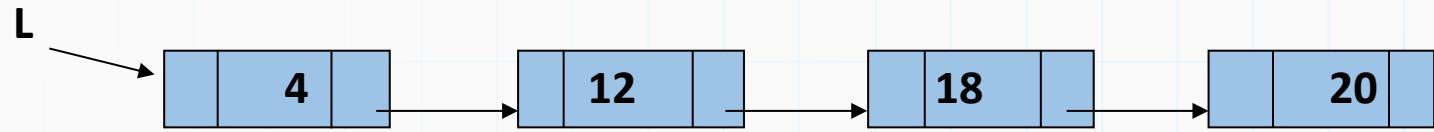


- Tamanho fixo, uma vez criada não pode aumentar nem diminuir 😞

# listas x vetores



Acesso:

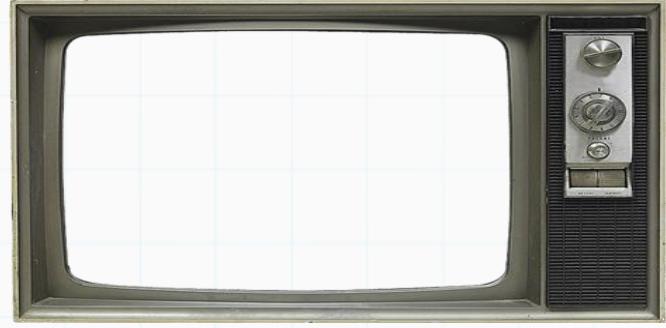


- Acesso sequencial ☹



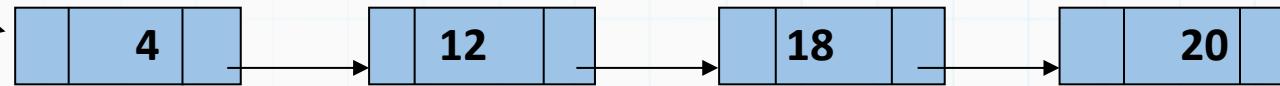
-Acesso imediato a posição indexada ☺

# listas x vetores



Inserir/Remover na posição 0:

L

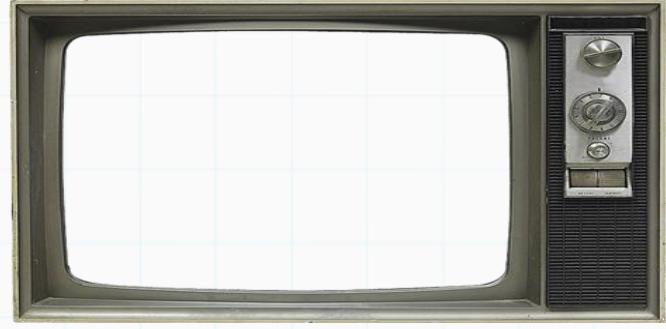


- tempo constante 😊



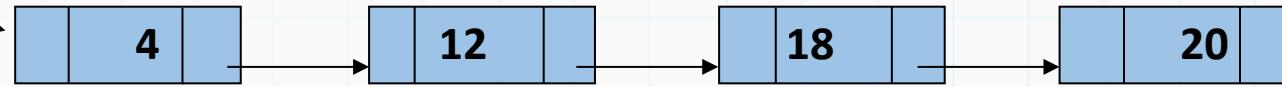
tem que empurrar os outros  $n$  elementos para frente/trás 😞

# listas x vetores



Inserir/Remover na posição n:

L

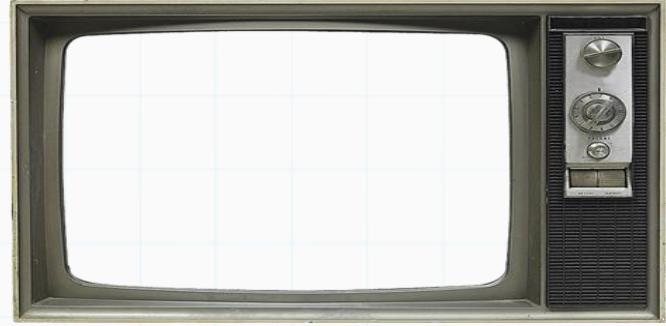


- tem que percorrer os outros n elementos 😞  
(lista linear)

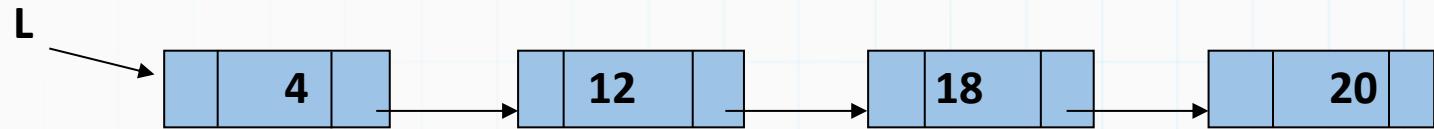


- tempo constante 😊

# listas x vetores



Busca:



Linear -> pior caso -> percorre  $n$  elementos  
Binária -> não pode pois precisa de indexação ☹

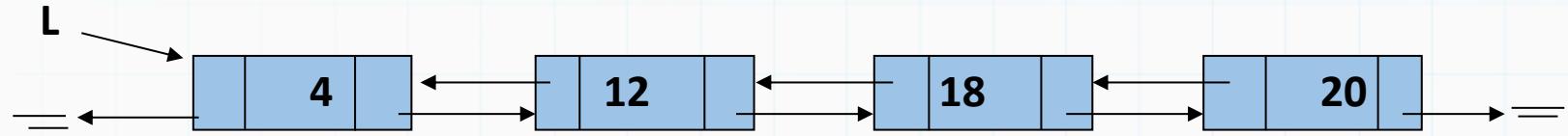


Linear -> pior caso -> percorre  $n$  elementos  
Binária -> percorre  $\log n$  elementos ☺

## Listas Duplamente Encadeadas



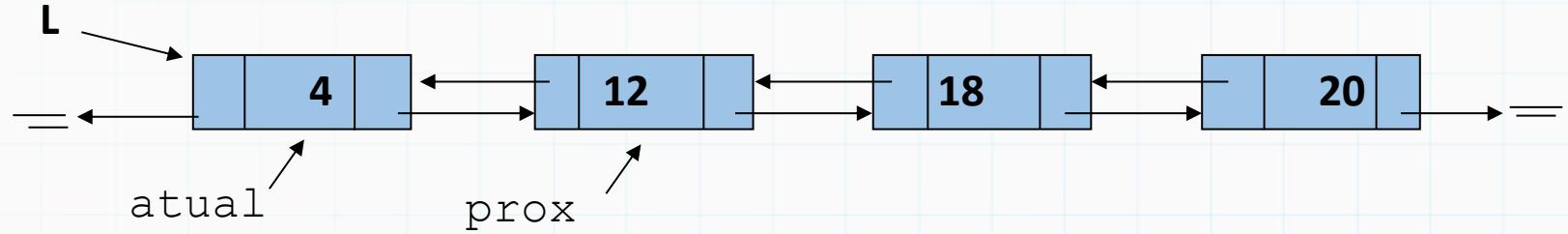
Exercício 2) Faça uma função que dado uma lista duplamente encadeada, inverta a lista através de reatribuição de ponteiros.



# Listas Duplamente Encadeadas



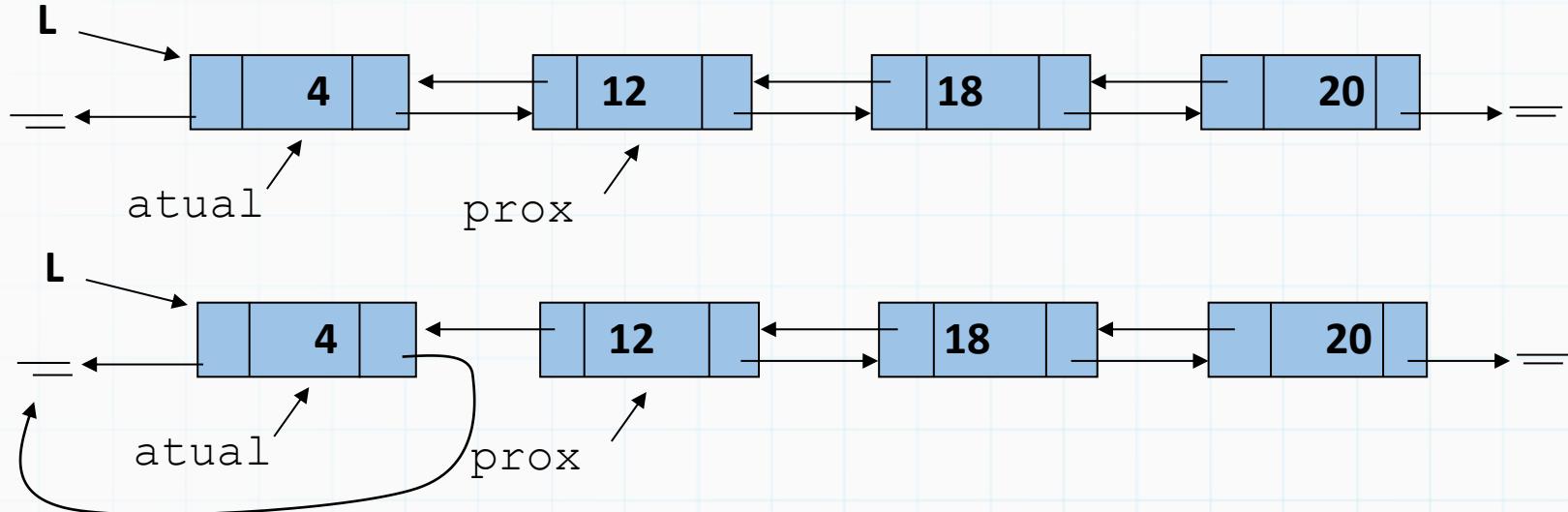
Exercício 2) Faça uma função que dado uma lista duplamente encadeada, inverta a lista através de re-atribuição de ponteiros.



# Listas Duplamente Encadeadas



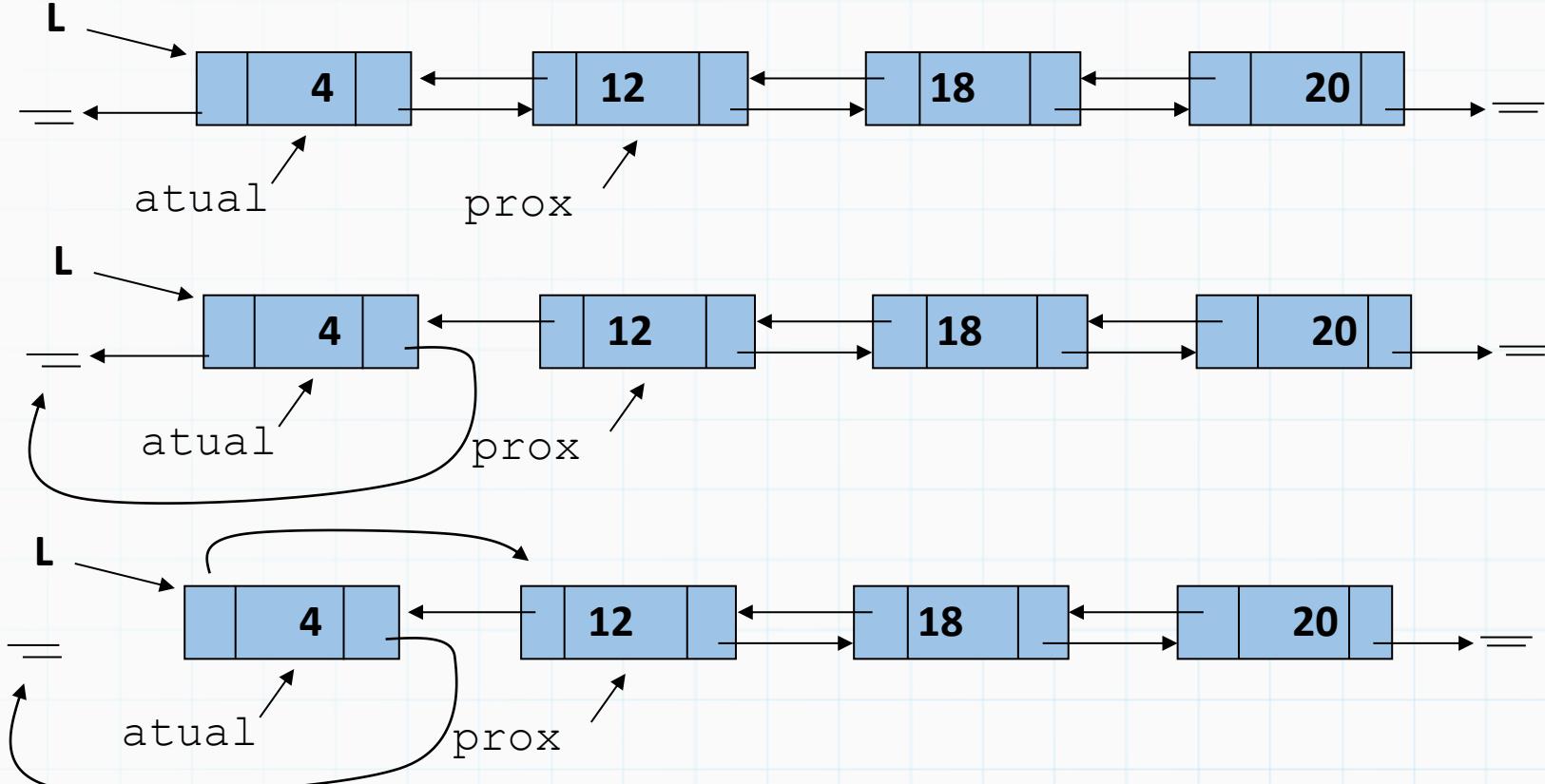
Exercício 2) Faça uma função que dado uma lista duplamente encadeada, inverta a lista através de re-atribuição de ponteiros.



# Listas Duplamente Encadeadas



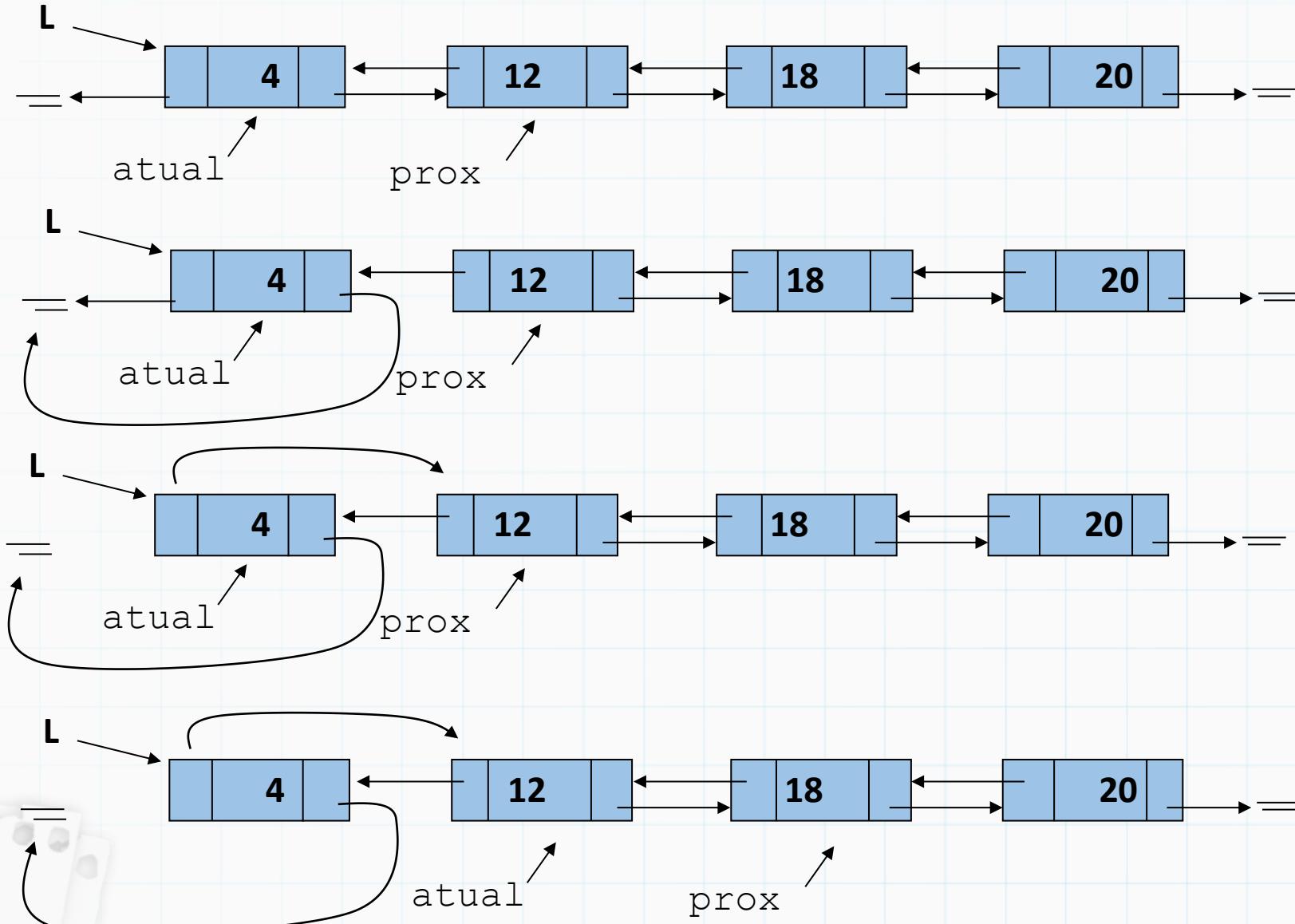
Exercício 2) Faça uma função que dado uma lista duplamente encadeada, inverta a lista através de re-atribuição de ponteiros.



# Listas Duplamente Encadeadas



Exercício 2) Faça uma função que dado uma lista duplamente encadeada, inverta a lista através de re-atribuição de ponteiros.



Não esqueça de:

- retornar a cabeça da lista
- tratar casos particulares (ex: lista vazia)

# Listas Duplamente Encadeadas



Exercício 2) Faça uma função que dado uma lista duplamente encadeada, inverta a lista através de re-atribuição de ponteiros.

```
lista* inverte_lista_C(lista* L)
{
    // caso a lista vazia ou apenas 1 elemento
    if (L == NULL || L->prox == NULL)
        return L;

    lista* atual = L, * proximo;
    while (atual != NULL)
    {
        proximo      = atual->prox;
        atual->prox  = atual->ant;
        atual->ant   = proximo;

        if (atual->prox != NULL)
            L = atual;

        atual = proximo;
    }

    return L;
}
```

```
lista* rearranjo (lista* l)
{
    lista *p, *q;
    int temp;

    if (l==NULL || l->prox==NULL)
        return;

    p = l;
    q = l->prox;
    while (q != NULL)
    {
        temp      = p->info;
        p->info  = q->info;
        q->info  = temp;
        p        = q->prox;

        if (p != NULL)
            q = p->prox;
        else
            q = NULL;
    }

    return l;
}
```



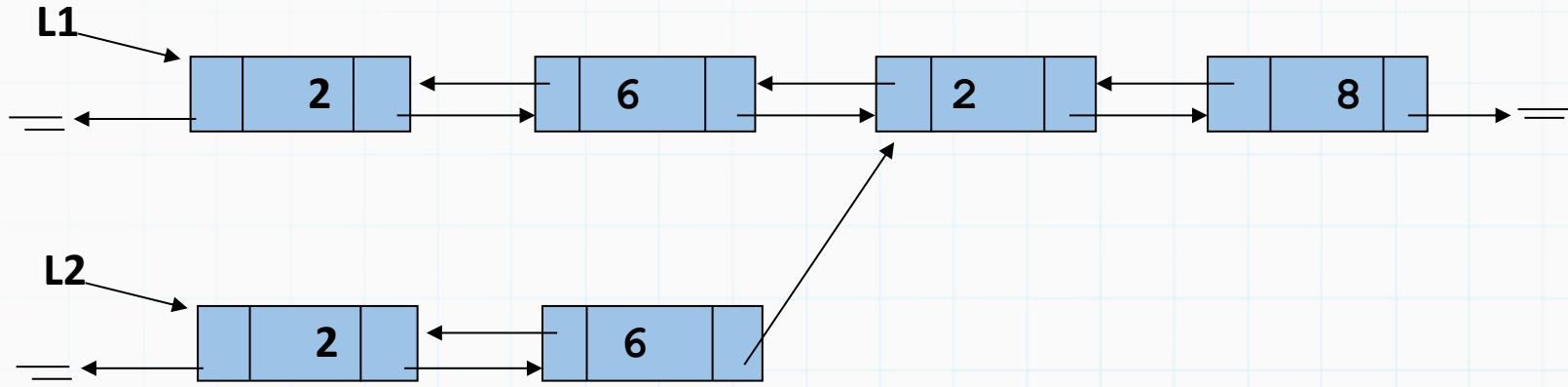
## listas

Fura olho: Dado uma lista simplesmente encadeada  $l = \{1, 2, 3, 4, 5, 6, 7\}$ , como ficaria a lista depois da chamada da função:

# Listas Duplamente Encadeadas



Exercício 3) Faça uma função para detecção de problemas de conexão entre listas. Dado duas lista duplamente encadeadas, detecte o erro, isto é ponto de conexão entre elas



OBS: os valores dos nós podem estar repetidos.

OBS2: retorne um ponteiro para o no da conexão, se não existir, retorne NULL

# Listas Duplamente Encadeadas

Exercício 3)

```
lista* conexao_D(lista* L1, lista* L2)
{
    if ((L1 == NULL) || (L2 == NULL))
        return NULL;

    lista* no1 = L1;
    while (no1 != NULL)
    {
        lista* no2 = L2;
        while (no2 != NULL)
        {
            if (no1 == no2)
                return no1;

            no2 = no2->prox;
        }

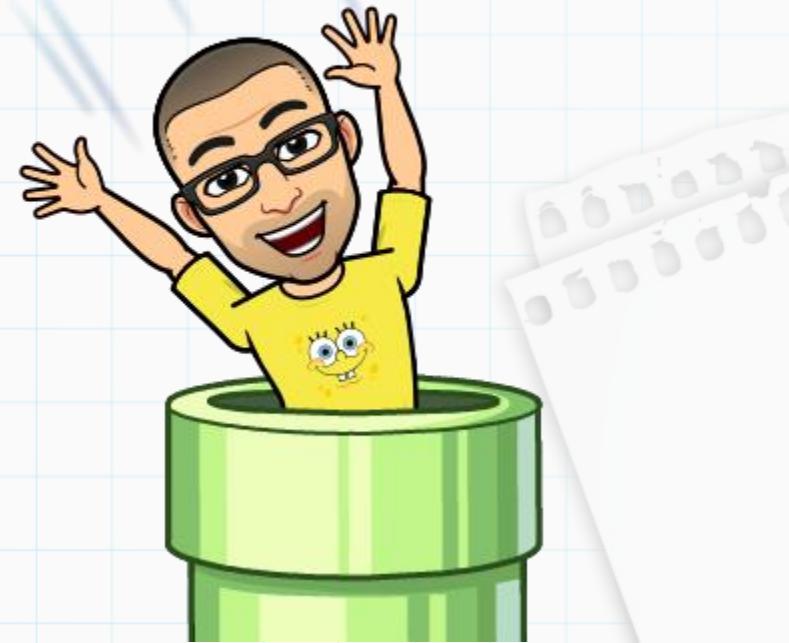
        no1 = no1->prox;
    }

    return NULL;
}
```

# Programação Estruturada



Até a próxima



Slides baseados no curso de Aline Nascimento