

Listas ligadas

Prof. Paulo Henrique Pisani

março/2022

Tópicos

- Listas com vetor/arranjo;
- Listas ligadas (parte 1):
 - Listas ligadas;
 - Algumas operações em listas ligadas.
- Listas ligadas (parte 2):
 - Listas ligadas com outros tipos de dados;
 - Listas duplamente ligadas;
 - Outros tipos: Listas com nó cabeça e Listas circulares.

Lista

- É uma estrutura de dados que permite as operações básicas a seguir:
 - Busca
 - Inserção
 - Remoção
- Vamos discutir duas formas de implementação:
 - Vetor/arranjo
 - Lista ligada

Listas com vetor/arranjo

- Itens dispostos em um arranjo sequencial;



Busca (lista com vetor)

- Por exemplo, buscar o item 30 na lista. Podemos realizar **busca sequencial**;
- É possível acessar elementos diretamente (com índice), o que pode permitir **busca binária (vetor ordenado)**.

10	20	30	40
----	----	----	----

Inserir (lista com vetor)

- Inserir um item pode precisar de deslocamentos;

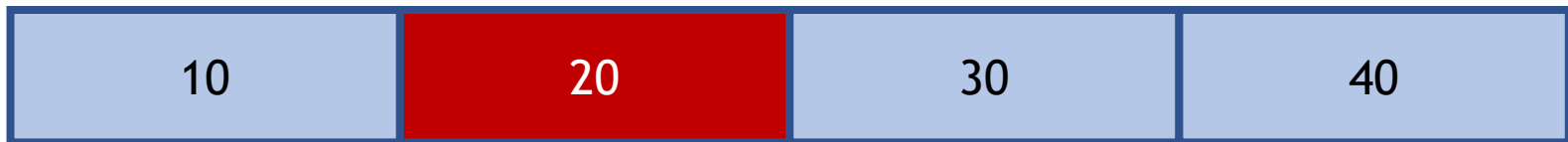


Inserir item 20 no índice 1, envolveu o deslocamento do 30.



Remover (lista com vetor)

- Remover um item pode precisar de deslocamentos;

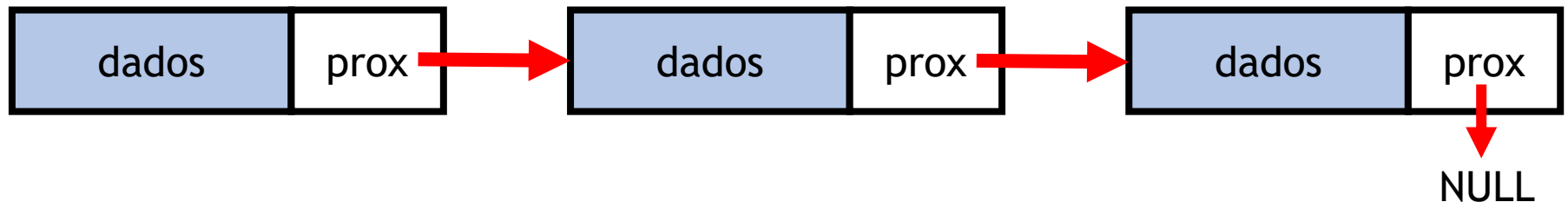


Remover item 20, envolveu o deslocamento do 30 e do 40.



Listas ligadas/encadeadas

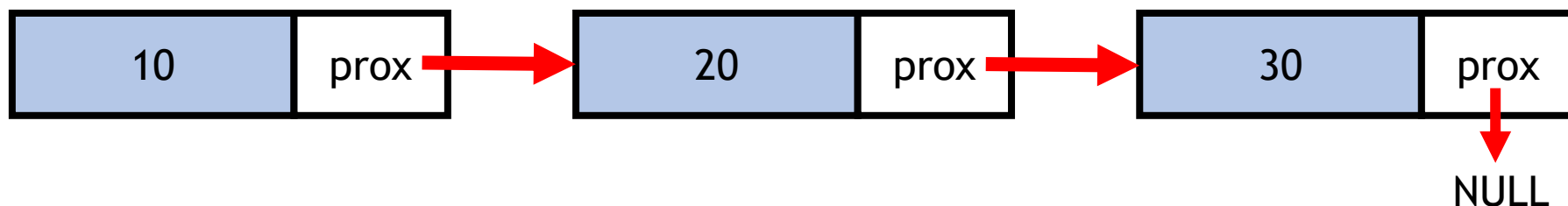
- Estrutura de dados que armazena os itens de forma não consecutiva na memória:
 - Cada item possui uma referência para o próximo.



Vantagens?
Desvantagens?

Buscar (lista ligada)

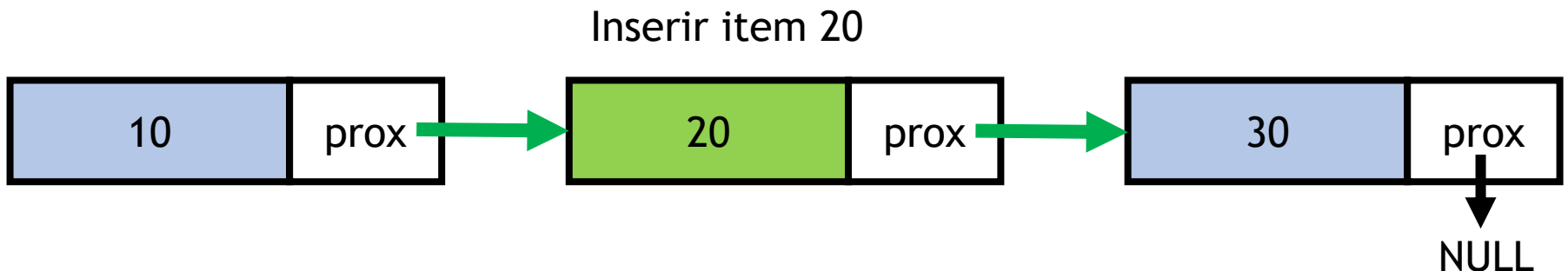
- Por exemplo, buscar o item 30 na lista. Podemos realizar **busca sequencial**;
- Acessar um item (por índice) requer percorrer a lista desde o início.



Por exemplo, acessar o terceiro item: Com a referência do início da lista, é necessário ir até o segundo para só depois acessar o terceiro.

Inserir (lista ligada)

- Não requer deslocamentos para inserção e remoção.

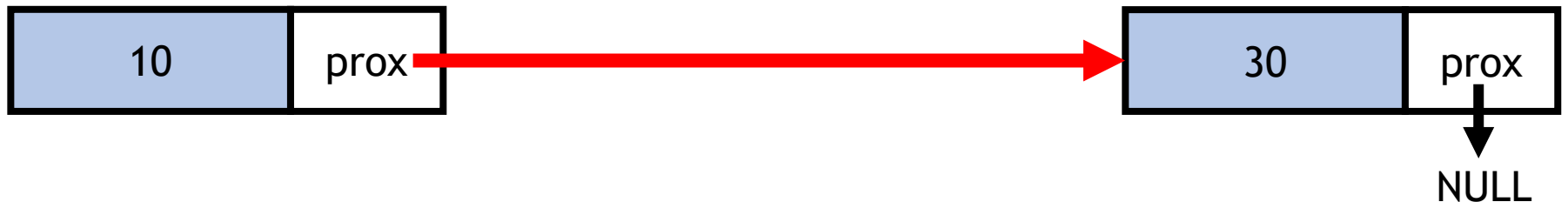


Remover (lista ligada)

- Não requer deslocamentos para inserção e remoção.



Remover item 20



Listas com vetor vs Listas ligadas

	Lista com vetor	Lista ligada
Operação: Busca	Permite acesso direto a um item (permite busca sequencial e binária)	Requer percorrer a lista para acessar um item (busca sequencial)
Operação: Inserção	Pode precisar de deslocamentos	Não requer deslocamentos
Operação: Remoção	Pode precisar de deslocamentos	Não requer deslocamentos
Uso de memória	Armazena apenas os itens	Requer armazenar ao menos uma referência junto com cada item

Ao definir qual implementação usar devemos avaliar a aplicação.

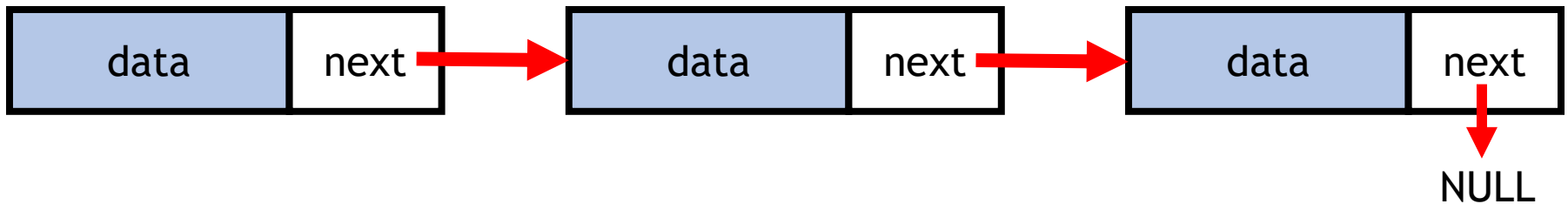
Listas ligadas/encadeadas

- Vários tipos:
 - Listas simplesmente ligadas (com e sem nó cabeça);
 - Listas duplamente ligadas (com e sem nó cabeça);
 - Listas circulares.

Listas ligadas

Listas ligadas

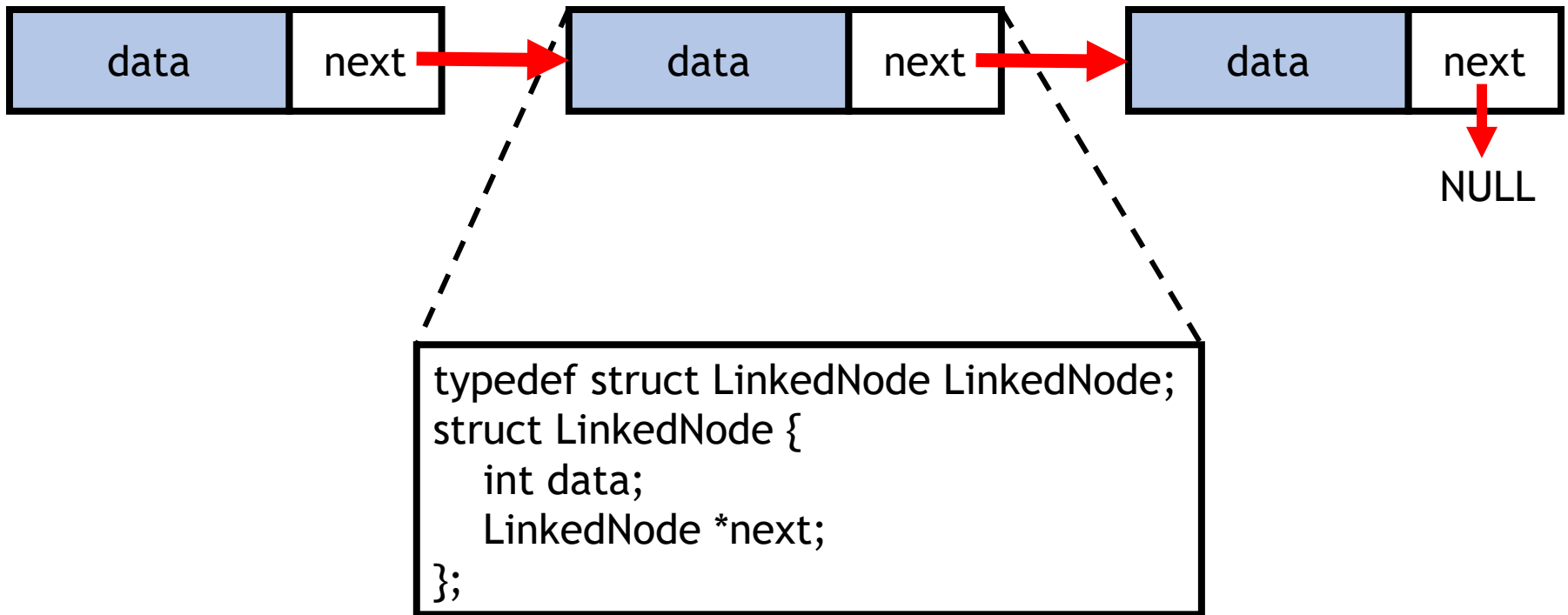
- Cada item é ligado ao próximo item.



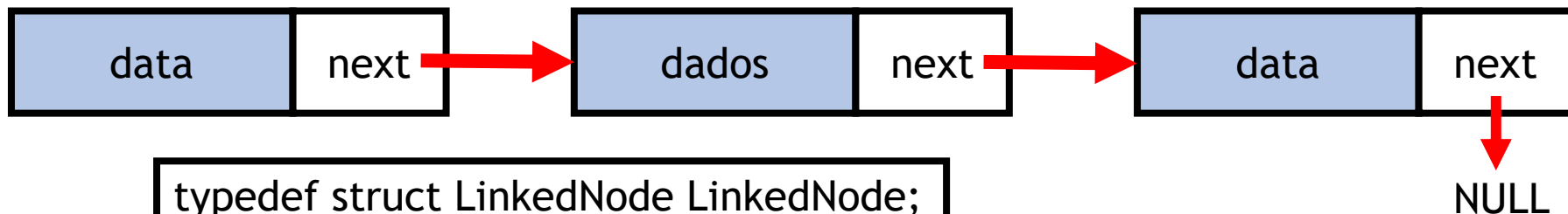
Como implementar
no C?

Listas ligadas

- Cada item é ligado ao próximo item.



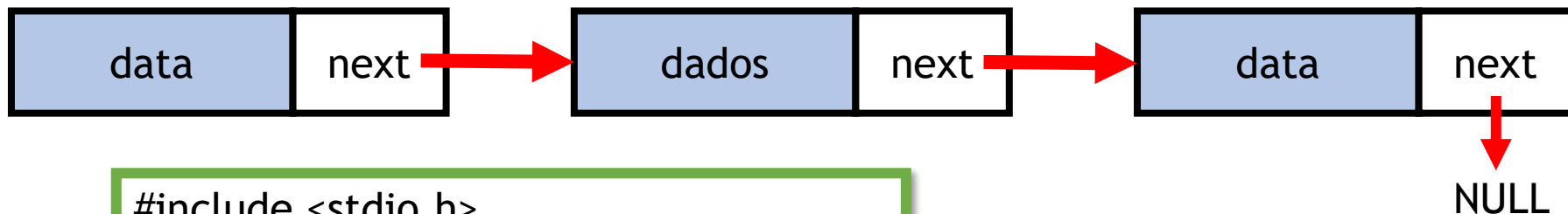
Listas ligadas



```
typedef struct ListNode ListNode;  
struct ListNode {  
    int data;  
    ListNode *next;  
};
```

```
typedef struct ListNode ListNode;  
struct ListNode {  
    int data;  
    ListNode *next;  
};
```

Listas ligadas



```
#include <stdio.h>
#include <stdlib.h>

typedef struct ListNode ListNode;
struct ListNode {
    int data;
    ListNode *next;
};

int main() {
    ListNode *inico = NULL;

    ...
}
```

O ponteiro para o primeiro item deve ser salvo.

Listas ligadas



```
#include <stdio.h>
#include <stdlib.h>

typedef struct ListNode ListNode;
struct ListNode {
    int data;
    ListNode *next;
};

int main() {
    ListNode *início = NULL;

    ...
}
```

O ponteiro para o primeiro item deve ser salvo.

Exemplo

- Alocar 3 nós na memória e formar uma lista ligada (definindo o valor do ponteiro next de cada nó);
- Imprimir os três nós (criar função que percorre uma lista de tamanho arbitrário).

Listas ligadas

- Vamos discutir algumas operações nos slides a seguir...

Algumas operações em listas ligadas

Inserir no final

Chamada:

```
inicio = inserir_final(inicio, 507);
```

```
LinkedList *inserir_final(LinkedList *inicio, int valor) {
```

Inserir nó no final da lista e retorna o novo início da lista:

- Novo início é atualizado apenas quando a lista é vazia;
- Nos demais casos, o novo início é o próprio valor do parâmetro inicio.

```
}
```

Inserir no final

Chamada:

```
inicio = inserir_final(inicio, 507);
```

```
LinkedNode *inserir_final(LinkedNode *inicio, int valor) {  
    LinkedNode *novo = malloc(sizeof(LinkedNode));  
    if (novo == NULL) return inicio;  
    novo->data = valor;  
    novo->next = NULL;  
  
    if (inicio == NULL) return novo;  
  
    LinkedNode *anterior = NULL;  
    LinkedNode *atual = inicio;  
    while (atual != NULL) {  
        anterior = atual;  
        atual = atual->next;  
    }  
  
    anterior->next = novo;  
    return inicio;  
}
```


Inserir no final

Chamada:

```
inicio = inserir_final(inicio, 507);
```

```
LinkedList *inserir_final(LinkedList *inicio, int valor) {
```

Aloca novo nó

```
    LinkedList *novo = malloc(sizeof(LinkedList));  
    if (novo == NULL) return inicio;  
    novo->data = valor;  
    novo->next = NULL;
```

Se a lista é vazia, → if (inicio == NULL) return novo;
o novo nó é o primeiro

Percorre lista,
anterior terá o
ponteiro para
último nó

```
    LinkedList *anterior = NULL;  
    LinkedList *atual = inicio;  
    while (atual != NULL) {  
        anterior = atual;  
        atual = atual->next;  
    }
```

next do último nó recebe o novo nó → anterior->next = novo;
return inicio;
}

Inserir no final

Chamada:

```
inicio = inserir_final_r(inicio, 507);
```

```
LinkedList *inserir_final_r(LinkedList *inicio, int valor) {
```

Veremos agora uma versão recursiva

```
}
```

Inserir no final

Chamada:

```
inicio = inserir_final_r(inicio, 507);
```

```
LinkedNode *inserir_final_r(LinkedNode *inicio, int valor) {
```

```
    if (inicio == NULL) {
```

Se a lista é
vazia, aloca nó
e retorna ele
como início

```
        LinkedNode *novo = malloc(sizeof(LinkedNode));
```

```
        if (novo == NULL) return inicio;
```

```
        novo->data = valor;
```

```
        novo->next = NULL;
```

```
        return novo;
```

```
    }
```

```
    inicio->next = inserir_final_r(inicio->next, valor);
```

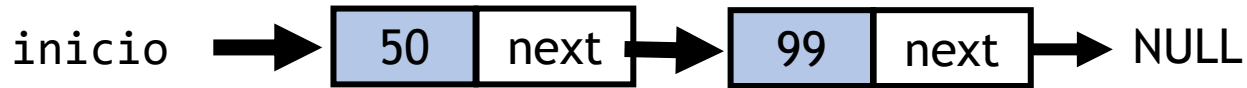
```
    return inicio;
```

```
}
```

Caso contrário, next
receberá o início da
lista ligada que inicia
em next, mas com o
novo valor inserido.

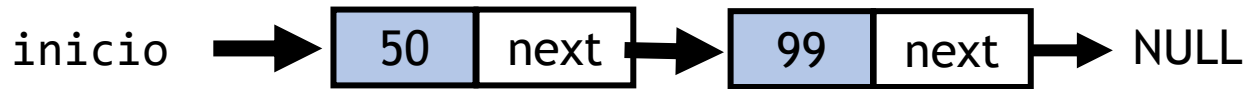
O início não é alterado, pois a lista não é vazia.

Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.



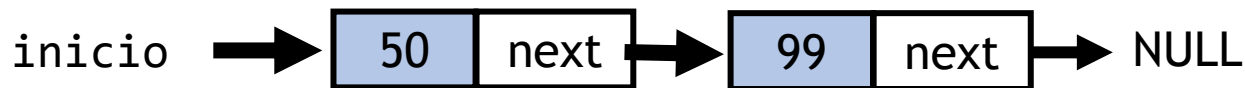
```
LinkedNode *inserir_final_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) {  
        LinkedNode *novo = malloc(sizeof(LinkedNode));  
        if (novo == NULL) return inicio;  
        novo->data = valor;  
        novo->next = NULL;  
        return novo;  
    }  
    inicio->next = inserir_final_r(inicio->next, valor);  
    return inicio;  
}
```

Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.



```
LinkedNode *inserir_final_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) {  
        LinkedNode *novo = malloc(sizeof(LinkedNode));  
        if (novo == NULL) return inicio;  
        novo->data = valor;  
        novo->next = NULL;  
        return novo;  
    }  
    inicio->next = inserir_final_r(inicio->next, valor);  
    return inicio;  
}
```

Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.



```
LinkedNode *inserir_final_r(LinkedNode *inicio, int valor) {
    if (inicio == NULL) {
        LinkedNode *novo = malloc(sizeof(LinkedNode));
        if (novo == NULL) return inicio;
        novo->data = valor;
        novo->next = NULL;
        return novo;
    }
```

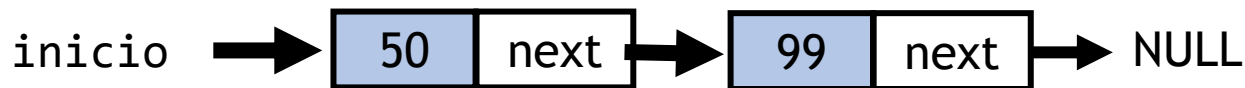
```
    inicio->next = inserir_final_r(inicio->next, valor);
```

```
    if (inicio == NULL) {
        LinkedNode *novo = malloc(sizeof(LinkedNode));
        if (novo == NULL) return inicio;
        novo->data = valor;
        novo->next = NULL;
        return novo;
    }
```

```
    inicio->next = inserir_final_r(inicio->next, valor);
    return inicio;
}
```

```
return inicio;
```

Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.



```
LinkedNode *inserir_final_r(LinkedNode *inicio, int valor) {
    if (inicio == NULL) {
        LinkedNode *novo = malloc(sizeof(LinkedNode));
        if (novo == NULL) return inicio;
        novo->data = valor;
        novo->next = NULL;
        return novo;
    }
```

```
    inicio->next = inserir_final_r(inicio->next, valor) {
        if (inicio == NULL) {
            LinkedNode *novo = malloc(sizeof(LinkedNode));
            if (novo == NULL) return inicio;
            novo->data = valor;
            novo->next = NULL;
            return novo;
        }
```

```
        inicio->next = inserir_final_r(inicio->next, valor) {
            if (inicio == NULL) {
                LinkedNode *novo = malloc(sizeof(LinkedNode));
                if (novo == NULL) return inicio;
                novo->data = valor;
                novo->next = NULL;
                return novo;
            }
            inicio->next = inserir_final_r(inicio->next, valor);
            return inicio;
        }
```

```
        return inicio;
    }
```

```
return inicio;
```

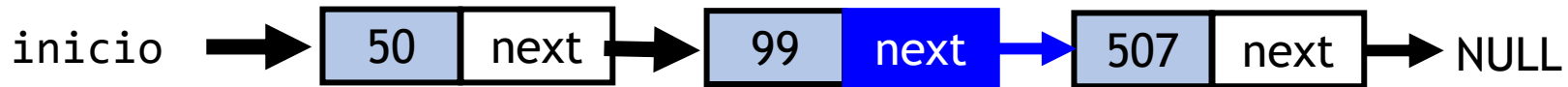
NULL

507

next

Aloca novo nó

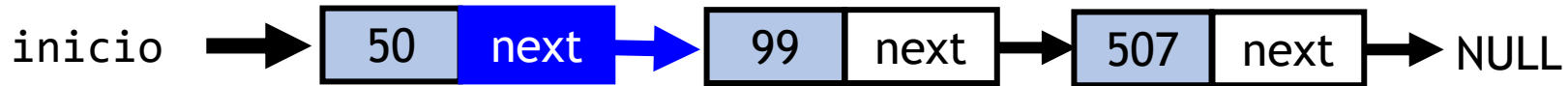
Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.



```
LinkedNode *inserir_final_r(Lista *l, int valor) {
    if (l->inicio == NULL) {
        LinkedNode *novo = malloc(sizeof(LinkedNode));
        if (novo == NULL) return l->inicio;
        novo->data = valor;
        novo->next = NULL;
        return novo;
    }
    l->inicio->next = inserir_final_r(l, valor);
    return l->inicio;
}
```

next de 99 aponta para o 507

Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.

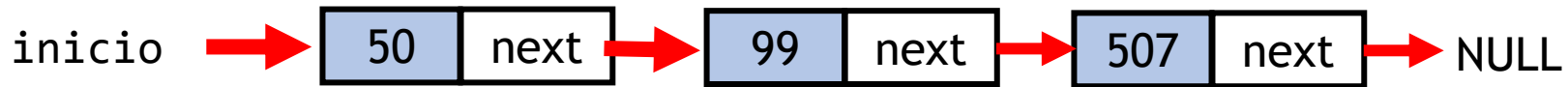


```
LinkedNode *inserir_final_r(Lista *l, int valor) {  
    if (inicio == NULL) {  
        LinkedNode *novo = malloc(sizeof(LinkedNode));  
        if (novo == NULL) return inicio;  
        novo->data = valor;  
        novo->next = NULL;  
        return novo;  
    }  
    inicio->next =  
    return inicio;  
}
```

next de 50 aponta para o 99

(o valor é o mesmo de antes, mas o valor é atribuído novamente)

Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.



- Observe que durante o processo de inserção, a lista foi reconstruída: todos os ponteiros next foram atualizados.
- Entretanto, o único que recebeu valor diferente foi o next do nó 99 (que possuía o valor NULL antes).

Inserir no final

- É possível otimizar a implementação do procedimento para inserir um nó no final da lista;
- Para isso, podemos sempre armazenar o ponteiro para o último item da lista; Dessa forma, não é necessário percorrer a lista até o final para inserir um novo item.

Remover

Chamada:

```
inicio = remover(inicio, 507);
```

```
LinkedList *remover(LinkedList *inicio, int valor) {
```

Procura nó com o valor informado, remove da lista e retorna o novo início da lista:

- Novo início é atualizado apenas quando o nó removido é o primeiro;
- Nos demais casos, o novo início é o próprio valor do parâmetro inicio.

```
}
```

Remove

Chamada:

```
inicio = remover(inicio, 507);
```

```
LinkedNode *remover(LinkedNode *inicio, int valor) {  
    LinkedNode *anterior = NULL;  
    LinkedNode *atual = inicio;  
    while (atual != NULL && atual->data != valor){  
        anterior = atual;  
        atual = atual->next;  
    }  
    if (atual != NULL) {  
        if (anterior != NULL)  
            anterior->next = atual->next;  
        else  
            inicio = atual->next;  
        free(atual);  
    }  
    return inicio;  
}
```

Procura nó

Ajusta ponteiros

Libera nó

Remover

Chamada:

```
inicio = remover_r(inicio, 507);
```

```
LinkedList *remover_r(LinkedList *inicio, int valor) {
```

Veremos agora uma versão recursiva

```
}
```

Remove

Chamada:

```
inicio = remover_r(inicio, 507);
```

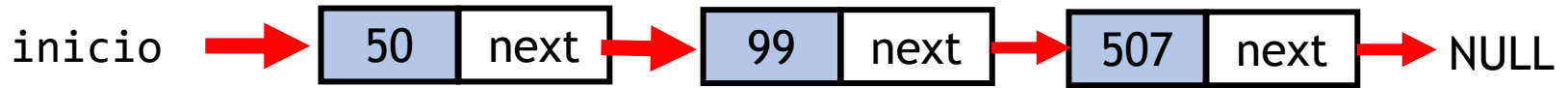
Se o início é o nó
a ser removido,
guarda o ponteiro
next (que será o
novo início) e
libera o nó.

```
LinkedList *remover_r(LinkedList *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        LinkedList *temp = inicio->next;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```

Caso contrário, next
receberá o início da
lista ligada que inicia
em next, mas com o
novo nó removido.

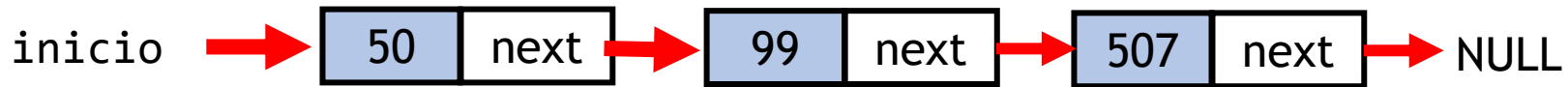
O início não é alterado.

Veremos o efeito nesta lista exemplo. Vamos remover o 99.



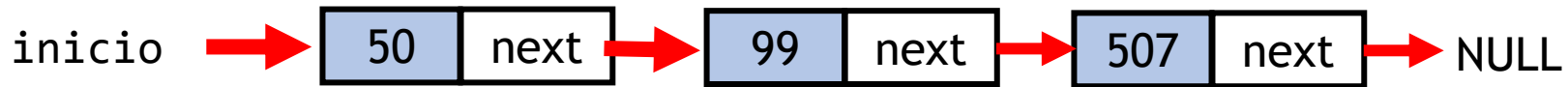
```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        LinkedNode *temp = inicio->next;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```


Veremos o efeito nesta lista exemplo. Vamos remover o 99.



```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {
    if (inicio == NULL) return NULL;
    if (inicio->data == valor) {
        LinkedNode *temp = inicio->next;
        free(inicio);
        return temp;
    }
    inicio->next = remover_r(inicio->next, valor);
    return inicio;
}
```

Veremos o efeito nesta lista exemplo. Vamos remover o 99.



```
LinkedNode *remover_r(Li
    50 | next int valor) {
    if (inicio == NULL) return NULL;
    if (inicio->data == valor) {
        LinkedNode *temp = inicio->next;
        free(inicio);
        return temp;
    }
    inicio->next =
        LinkedNode *remover_r(Li
            99 | next int valor) {
                if (inicio == NULL) return NULL;
                if (inicio->data == valor) {
                    LinkedNode *temp =
                        507 | next
                    free(inicio);
                    return temp;
                }
                inicio->next = remover_r(inicio->next, valor);
                return inicio;
            }

    return inicio;
}
```

Veremos o efeito nesta lista exemplo. Vamos remover o 99.



```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        LinkedNode *temp = inicio->next;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```

50	next
----	------

```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        LinkedNode *temp = inicio->next;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```

99	next
----	------

```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        LinkedNode *temp = inicio->next;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```

507	next
-----	------

Veremos o efeito nesta lista exemplo. Vamos remover o 99.



```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        LinkedNode *temp = inicio->next;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```

50	next
----	------

```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        LinkedNode *temp = inicio->next;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```

99	next
----	------

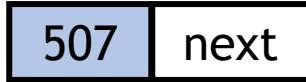
```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        LinkedNode *temp = inicio->next;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```

507	next
-----	------

Veremos o efeito nesta lista exemplo. Vamos remover o 99.



```
LinkedNode *remover_r(List *li, int valor) {  
    if (li == NULL) return NULL;  
    if (li->data == valor) {  
        LinkedNode *temp = li->next;  
        free(li);  
        return temp;  
    }  
    li->next =  
    return li;  
}
```



next de 50 aponta para o 507

Veremos o efeito nesta lista exemplo. Vamos remover o 99.



Liberar lista ligada



Como implementar
no C?

Liberar lista ligada

Chamada:

```
liberar_lista(inicio);
```

```
void liberar_lista(LinkedNode *inicio) {  
    if (inicio == NULL) return;  
    liberar_lista(inicio->next);  
    free(inicio);  
}
```


Exercícios

- Escreva funções em C para realizar as seguintes operações com listas ligadas:
 1. Concatenar duas listas;
 2. Inverter uma lista sobre ela mesma (sem criar uma nova);
 3. Dividir uma lista em duas metades. Se o tamanho da lista é ímpar, a segunda metade terá tamanho ímpar;
 4. Eliminar o primeiro item de uma lista;
 5. Eliminar o último item de uma lista;
 6. Inserir um item na posição i da lista;
 7. Remover o item da posição i da lista.

Exercício 1

- Escrever um programa que fique lendo números em uma lista ligada até que o usuário digite o número -1;
- Depois imprima a lista de números.

Exercício 2

- Adapte o exercício anterior de forma que os elementos sejam inseridos em ordem crescente na lista ligada.

Listas ligadas com
outros tipos de dados

Estrutura do nó

- Vimos a implementação de listas ligadas para um nó que armazena um número inteiro:

```
typedef struct ListNode ListNode;  
struct ListNode {  
    int data;  
    ListNode *next;  
};
```

Estrutura do nó

- Podemos utilizar outro typedef para o tipo de dados do nó; Nesse caso, as funções também usariam esse tipo:

```
typedef int TIPO;
```

```
typedef struct ListNode ListNode;  
struct ListNode {  
    TIPO data;  
    ListNode *next;  
};
```

```
ListNode *inserir_final(ListNode *inicio, TIPO valor) {  
    ...  
}
```

Estrutura do nó

- Além de trocar para float ou double mais facilmente, também poderíamos trocar para um ponteiro de char (pode ser usado para strings):

```
typedef char* TIPO;
```

```
typedef struct ListNode ListNode;  
struct ListNode {  
    TIPO data;  
    ListNode *next;  
};
```

```
ListNode *inserir_final(ListNode *inicio, TIPO valor) {  
    ...  
}
```

Estrutura do nó

- Esse tipo inclusive ser outra estrutura ou ponteiro para outra estrutura:

```
struct Pedido {  
    int codigo;  
    char *descricao;  
};
```

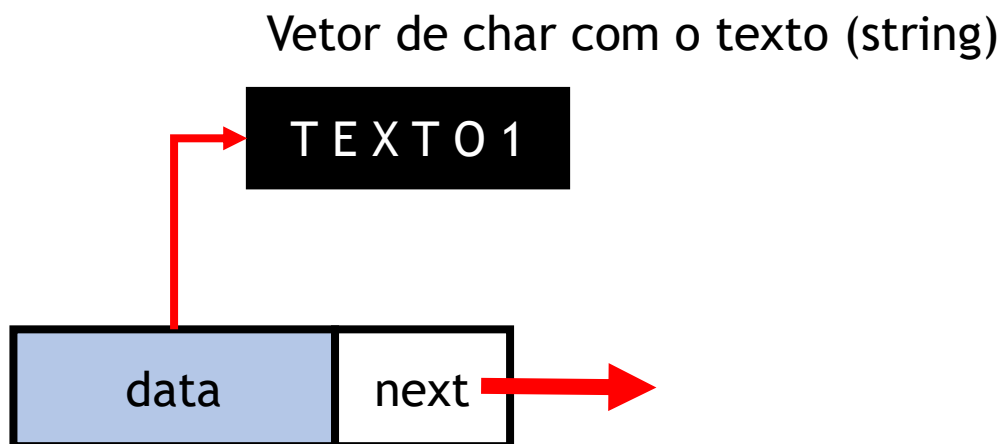
```
typedef struct Pedido TIPO;
```

```
typedef struct ListNode ListNode;  
struct ListNode {  
    TIPO data;  
    ListNode *next;  
};
```


Exemplo

- Escrever um programa que leia uma lista de strings e armazene em uma lista ligada.

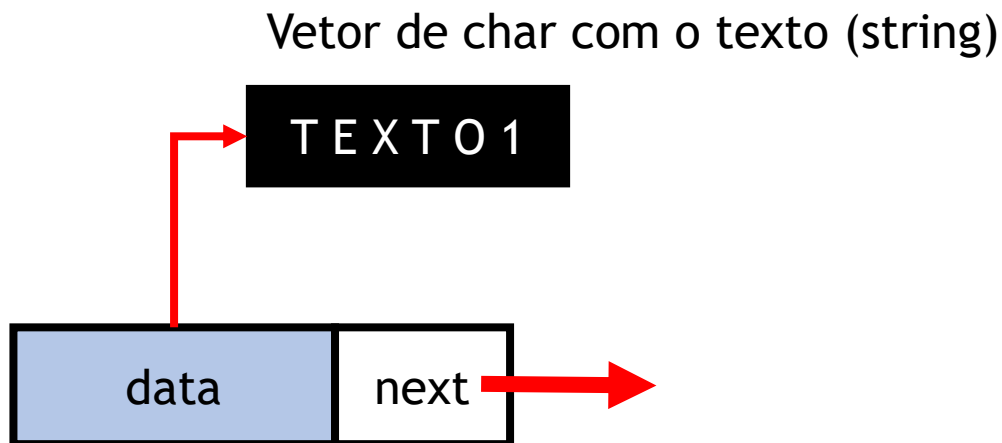
Estrutura da lista de strings



Cada nó armazena dois ponteiros apenas:

- data: um ponteiro para o vetor de char com o texto (string);
- next: um ponteiro para o próximo elemento.

Estrutura da lista de strings

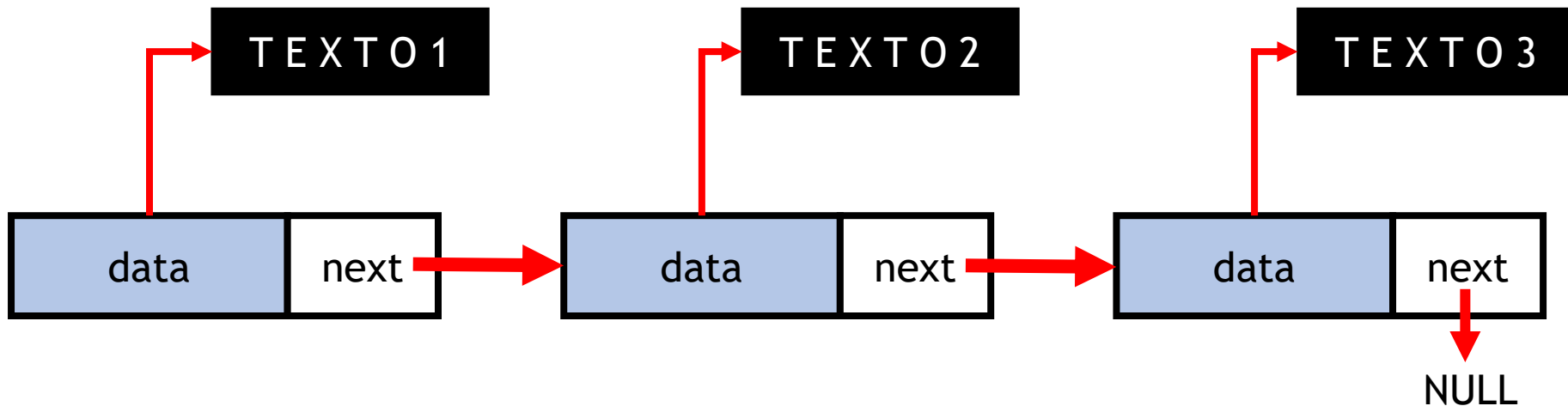


Cada nó armazena dois ponteiros apenas:

- data: um ponteiro para o vetor de char com o texto (string);
- next: um ponteiro para o próximo elemento.

Importante: ao liberar o nó da memória, é necessário liberar o vetor de char com o texto também.

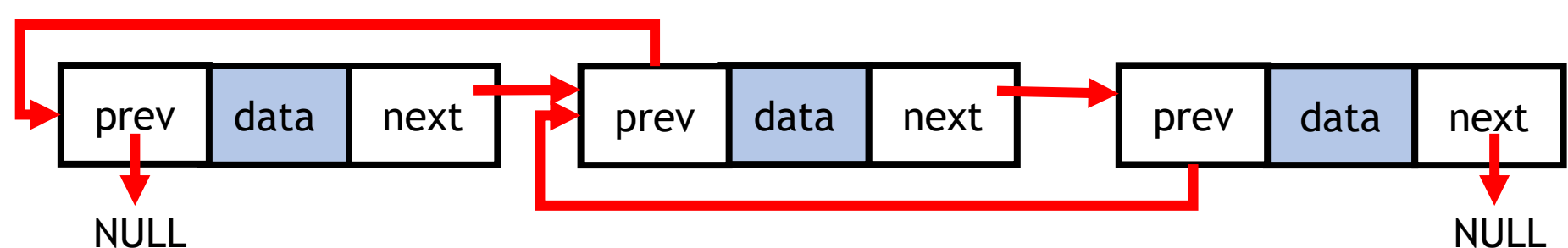
Estrutura da lista de strings



Listas duplamente
ligadas

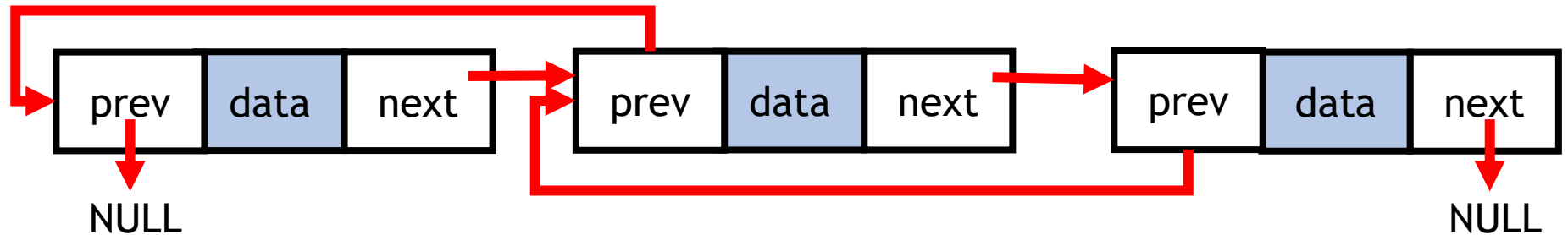
Listas duplamente ligadas

- Cada item é ligado ao próximo item e também ao anterior;
- **Vantagem: a lista pode ser percorrida em ambas as direções.**



Listas duplamente ligadas

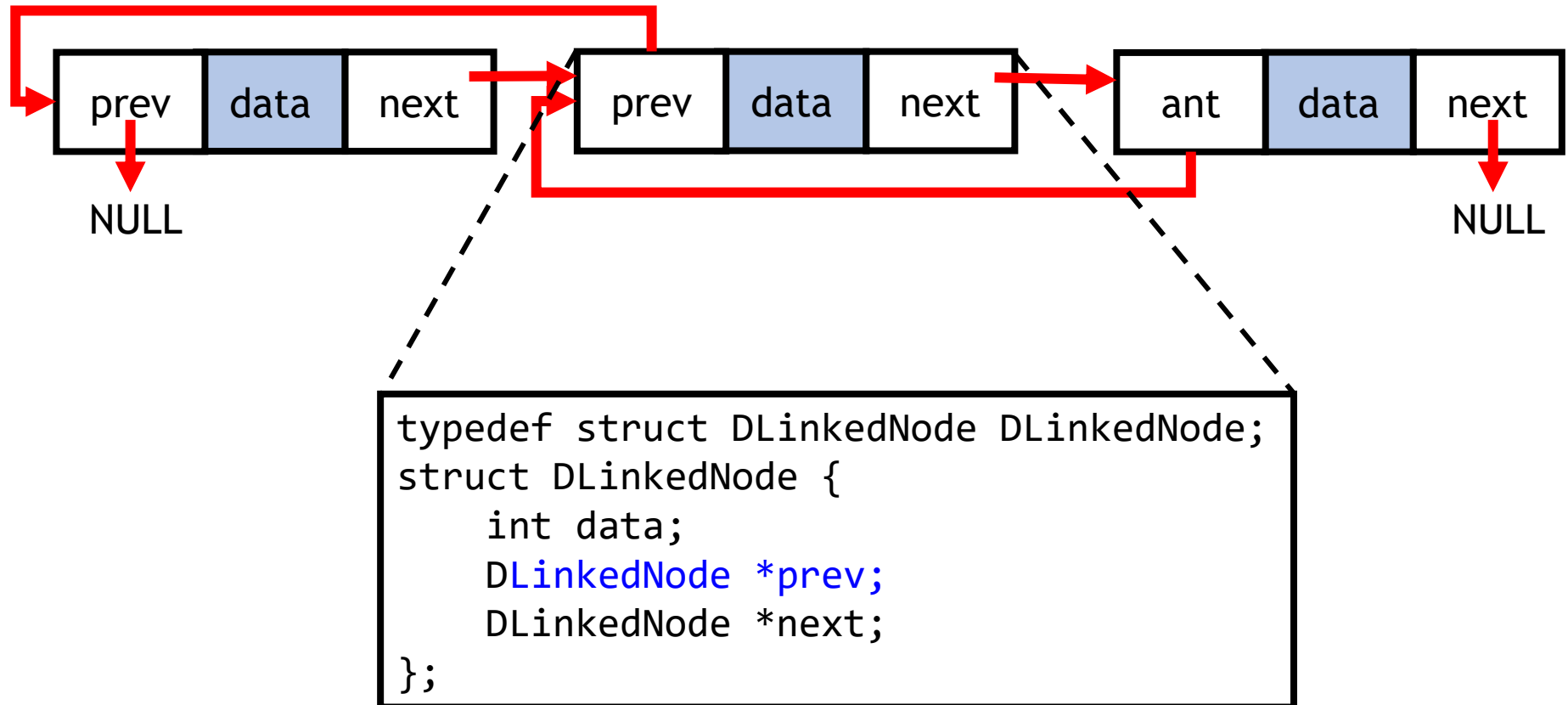
- Cada item é ligado ao próximo item e também ao anterior;



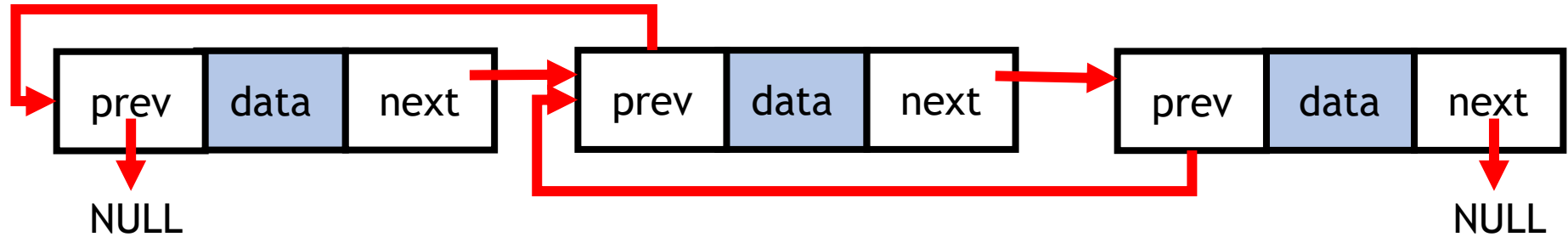
Como implementar
no C?

Listas duplamente ligadas

- Cada item é ligado ao próximo item e também ao anterior;



Listas duplamente ligadas

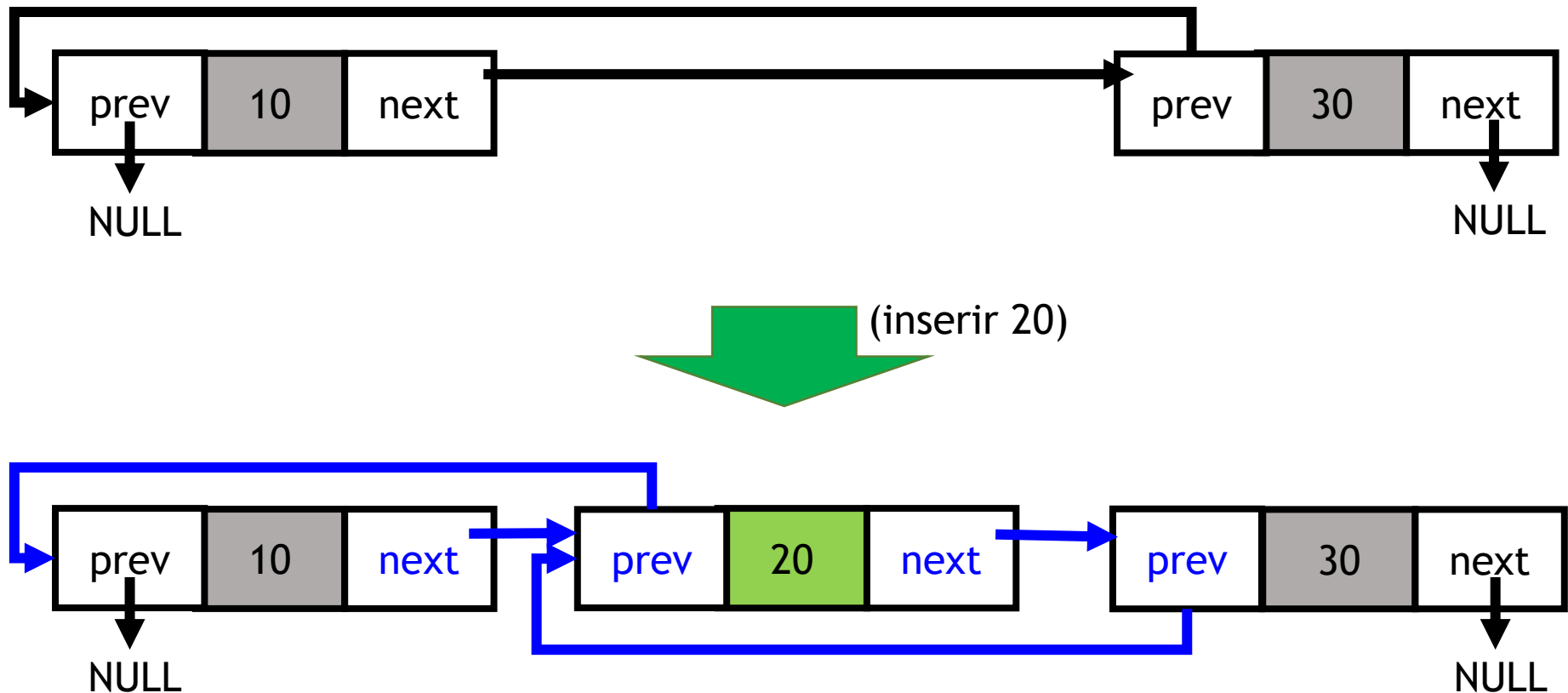


```
typedef struct DLinkedListNode DLinkedListNode;  
struct DLinkedListNode {  
    int data;  
    DLinkedListNode *prev;  
    DLinkedListNode *next;  
};
```

```
typedef struct DLinkedListNode DLinkedListNode;  
struct DLinkedListNode {  
    int data;  
    DLinkedListNode *prev;  
    DLinkedListNode *next;  
};
```

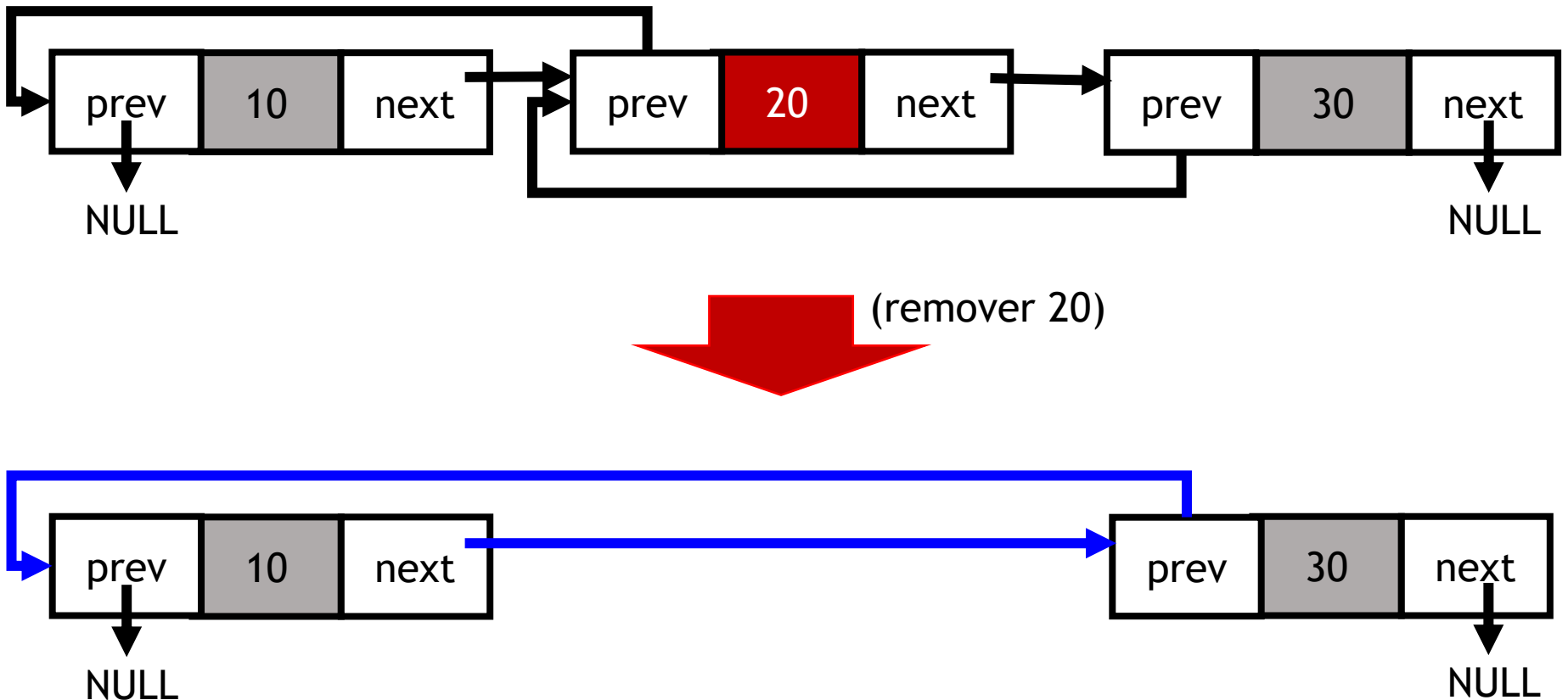
Listas duplamente ligadas

- Inserção de item:



Listas duplamente ligadas

- Remoção de item:



Algumas operações em listas duplamente ligadas

Operações em listas duplamente ligadas

- O processo é similar ao das listas ligadas que vimos (com apenas um ponteiro *next*);
- Com listas duplamente ligadas, ao alterar a lista, é necessário atualizar o ponteiro *prev* também.

Inserir no final

Chamada:

```
inicio = inserir_final(inicio, 507);
```

```
DLinkedList *inserir_final(DLinkedList *inicio, int valor) {
```

Insere nó no final da lista e retorna o novo início da lista:

- Novo início é atualizado apenas quando a lista é vazia;
- Nos demais casos, o novo início é o próprio valor do parâmetro inicio.

```
}
```

Inserir no final

Chamada:

```
inicio = inserir_final(inicio, 507);
```

```
DLinkedListNode *inserir_final(DLinkedListNode *inicio, int valor) {
    DLinkedListNode *novo = malloc(sizeof(DLinkedListNode));
    if (novo == NULL) return inicio;
    novo->data = valor;
    novo->prev = NULL;
    novo->next = NULL;

    if (inicio == NULL) return novo;

    DLinkedListNode *anterior = NULL;
    DLinkedListNode *atual = inicio;
    while (atual != NULL) {
        anterior = atual;
        atual = atual->next;
    }
    novo->prev = anterior;
    anterior->next = novo;
    return inicio;
}
```

Inserir no final

- É possível otimizar a implementação do procedimento para inserir um nó no final da lista;
- Para isso, podemos sempre armazenar o ponteiro para o último item da lista; Dessa forma, não é necessário percorrer a lista até o final para inserir um novo item.

Remover

Chamada:

```
inicio = remover(inicio, 507);
```

```
DLinkedList *remover(DLinkedList *inicio, int valor) {
```

Procura nó com o valor informado, remove da lista e retorna o novo início da lista:

- Novo início é atualizado apenas quando o nó removido é o primeiro;
- Nos demais casos, o novo início é o próprio valor do parâmetro inicio.

```
}
```

Remover

Chamada:

```
inicio = remover(inicio, 507);
```

```
DLinkedListNode *remover(DLinkedListNode *inicio, int valor) {  
    DLinkedListNode *atual = inicio;  
    while (atual != NULL && atual->data != valor)  
        atual = atual->next;  
    if (atual != NULL) {  
        DLinkedListNode *anterior = atual->prev;  
        DLinkedListNode *proximo = atual->next;  
        if (anterior != NULL)  
            anterior->next = proximo;  
        else  
            inicio = proximo;  
        if (proximo != NULL)  
            proximo->prev = anterior;  
        free(atual);  
    }  
    return inicio;  
}
```

Exemplo

- Criar uma estrutura para lista ligada que armazene ponteiros para o primeiro item e para o último item.

```
typedef struct ListaLigada ListaLigada;  
struct ListaLigada {  
    DLinkedList *inicio, *fim;  
};
```

- Modificar as funções `inserir_final` e `remover` para considerar esta nova estrutura.
 - Com o ponteiro para o último item, a função `inserir_final` não precisará mais percorrer toda a lista sempre que um novo item for adicionado.

Inserir e remover

- Vimos a implementação das funções inserir no final e remover;
- Como seria a versão recursiva dessas duas funções?

Inserir no final

Chamada:

```
inicio = inserir_final_r(inicio, 507);
```

```
DLinkedListNode *inserir_final_r(DLinkedListNode *inicio, int valor) {  
    if (inicio == NULL) {  
        DLinkedListNode *novo = malloc(sizeof(DLinkedListNode));  
        if (novo == NULL) return inicio;  
        novo->data = valor;  
        novo->prev = NULL;  
        novo->next = NULL;  
        return novo;  
    }  
    inicio->next = inserir_final_r(inicio->next, valor);  
    inicio->next->prev = inicio;  
    return inicio;  
}
```

Remover

Chamada:

```
inicio = remover_r(inicio, 507);
```

```
DLinkedListNode *remover_r(DLinkedListNode *inicio, int valor) {  
    if (inicio == NULL) return NULL;  
    if (inicio->data == valor) {  
        DLinkedListNode *temp = inicio->next;  
        if (inicio->next != NULL) inicio->next->prev = inicio->prev;  
        free(inicio);  
        return temp;  
    }  
    inicio->next = remover_r(inicio->next, valor);  
    return inicio;  
}
```

Estrutura do nó

- Vimos a implementação de listas duplamente ligadas para um nó que armazena um número inteiro:

```
typedef struct DLinkedNode DLinkedNode;  
struct DLinkedNode {  
    int data;  
    DLinkedNode *prev;  
    DLinkedNode *next;  
};
```

Estrutura do nó

- Mas para a lista duplamente ligada também podemos usar um typedef na definição do tipo.

```
typedef int TIPO;
```

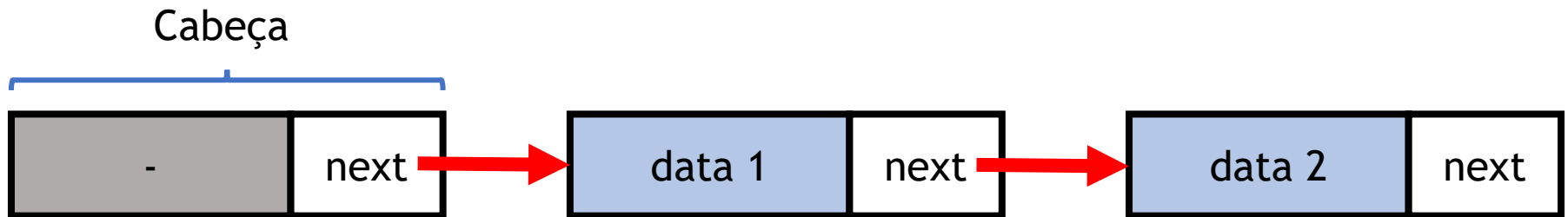
```
typedef struct DLinkedNode DLinkedNode;  
struct DLinkedNode {  
    TIPO data;  
    DLinkedNode *prev;  
    DLinkedNode *next;  
};
```


Outros tipos

Com nó cabeça e lista circular

Listas simplesmente ligadas com nó cabeça

- Cada item é ligado somente ao próximo item;
- O primeiro item não armazena itens da lista (e nunca é excluído);
- **Vantagem:** não é necessário verificar se a lista está vazia (o item cabeça nunca é removido).



Listas circulares

- Cada item é ligado somente ao próximo item e o último item é ligado ao primeiro.



Resumo

Listas

```
graph LR; A((Listas)) --> B[Listas com arranjos]; A --> C[Listas ligadas/encadeadas/enlaçadas];
```

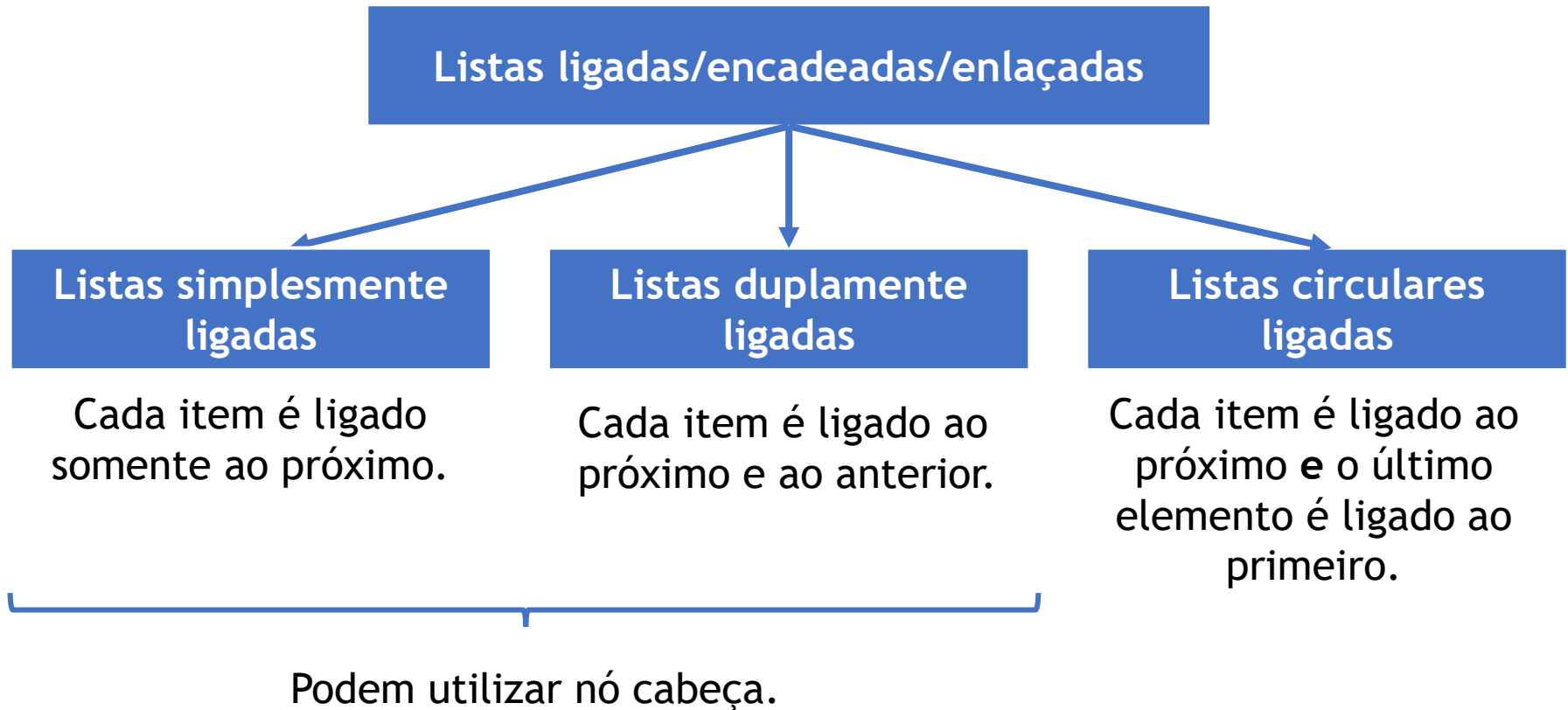
Listas com arranjos

- Simples para usar
- Alocação em bloco contínuo
- Acesso a um item em tempo constante
- Requer saber a quantidade de itens previamente (para alocação)
- Inserção/Remoção requer deslocamentos
- Expansão custosa (realocar e copiar)

Listas ligadas/encadeadas/enlaçadas

- Não requer conhecer a quantidade de itens previamente
- Inserção e remoção não requer deslocamentos
- Acesso a uma posição necessita percorrer a lista
- Memória extra para os ponteiros

Resumo



Referências

- Nivio Ziviani. Projeto de Algoritmos: com implementações em Pascal e C. Cengage Learning, 2015.
- Jayme L. Szwarcfiter, Lilian Markenzon. Estruturas de Dados e Seus Algoritmos. 3ª edição. LTC, 2012.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. Elsevier, 2012.