Listas ligadas (parte 2)

Prof. Paulo Henrique Pisani

novembro/2021

Tópicos

- Listas ligadas (parte 2):
  - Listas ligadas com outros tipos de dados;
  - Listas duplamente ligadas;
  - Outros tipos: Listas com nó cabeça e Listas circulares.

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 LinkedNode LinkedNode;
struct LinkedNode {
    int data;
    LinkedNode *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 LinkedNode LinkedNode;
struct LinkedNode {
    TIPO data;
    LinkedNode *next;
};

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

Estrutura do nó

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

```
typedef char* TIPO;

typedef struct LinkedNode LinkedNode;
struct LinkedNode {
    TIPO data;
    LinkedNode *next;
};

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

Estrutura do nó

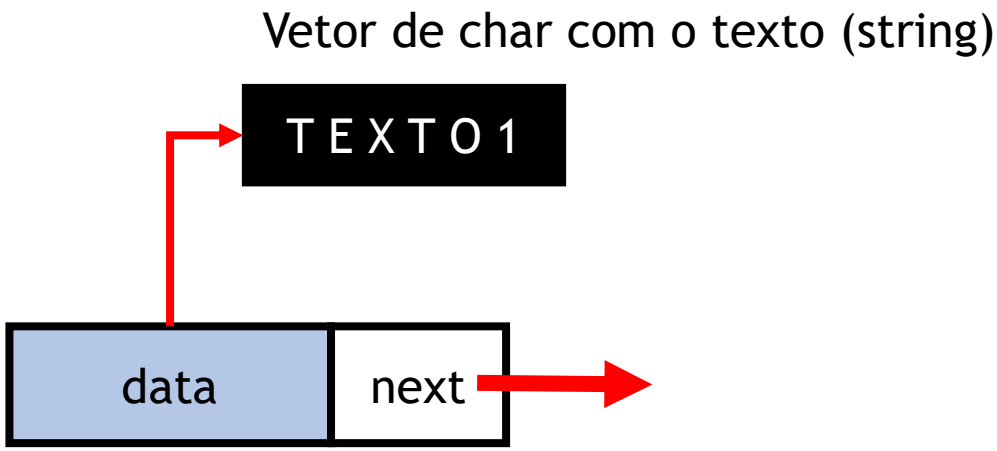
- Esse tipo inclusive ser outra estrutura ou ponteiro para outra estrutura:

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

typedef struct Pedido TIPO;

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

Exemplo

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

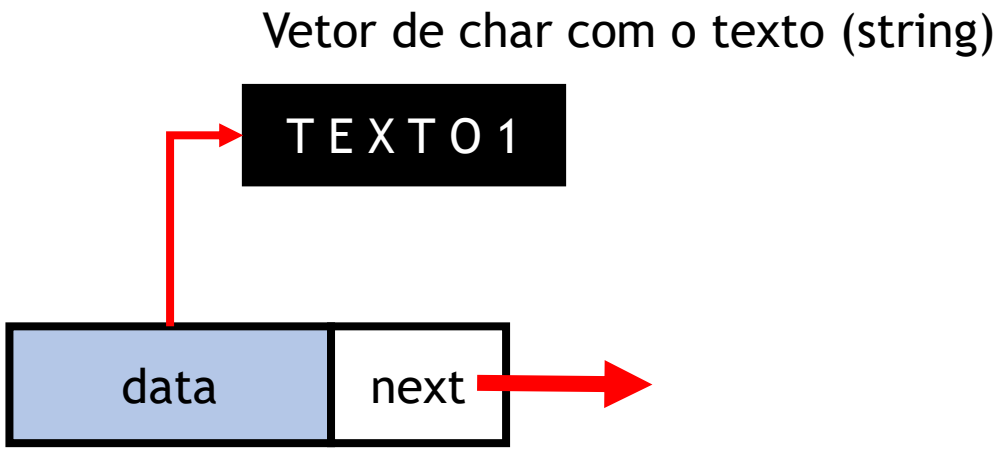
Estrutura da lista de strings



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

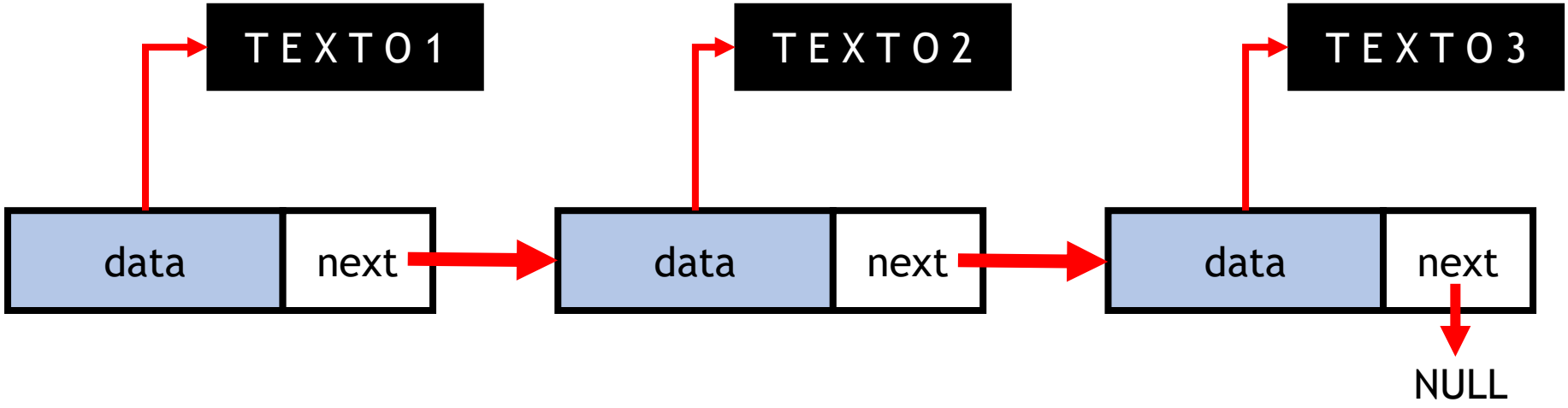


Cada nó armazena dois ponteiros apenas:

- 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.

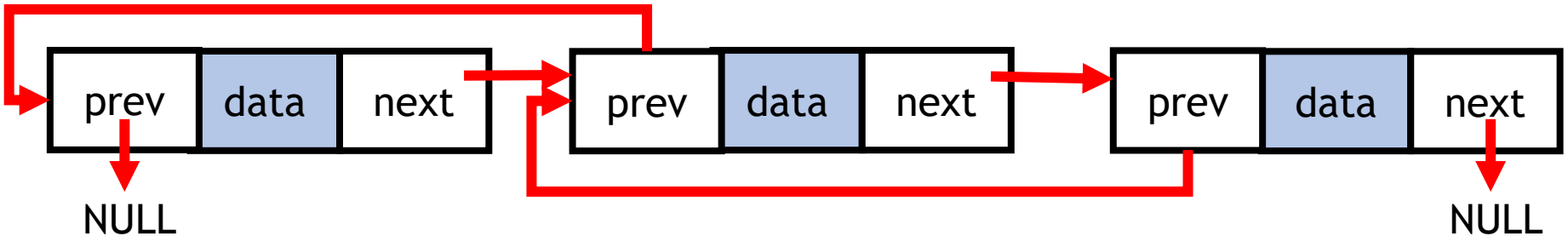
Estrutura da lista de strings



Listas duplamente ligadas

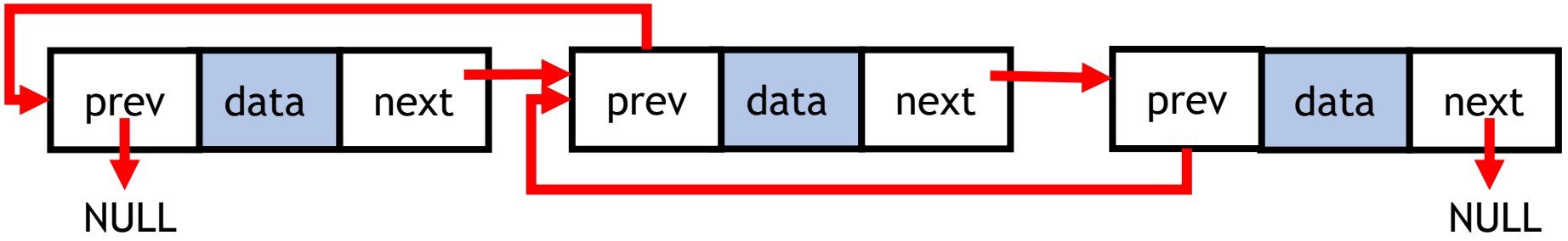
Listas duplamente ligadas

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



Listas duplamente ligadas

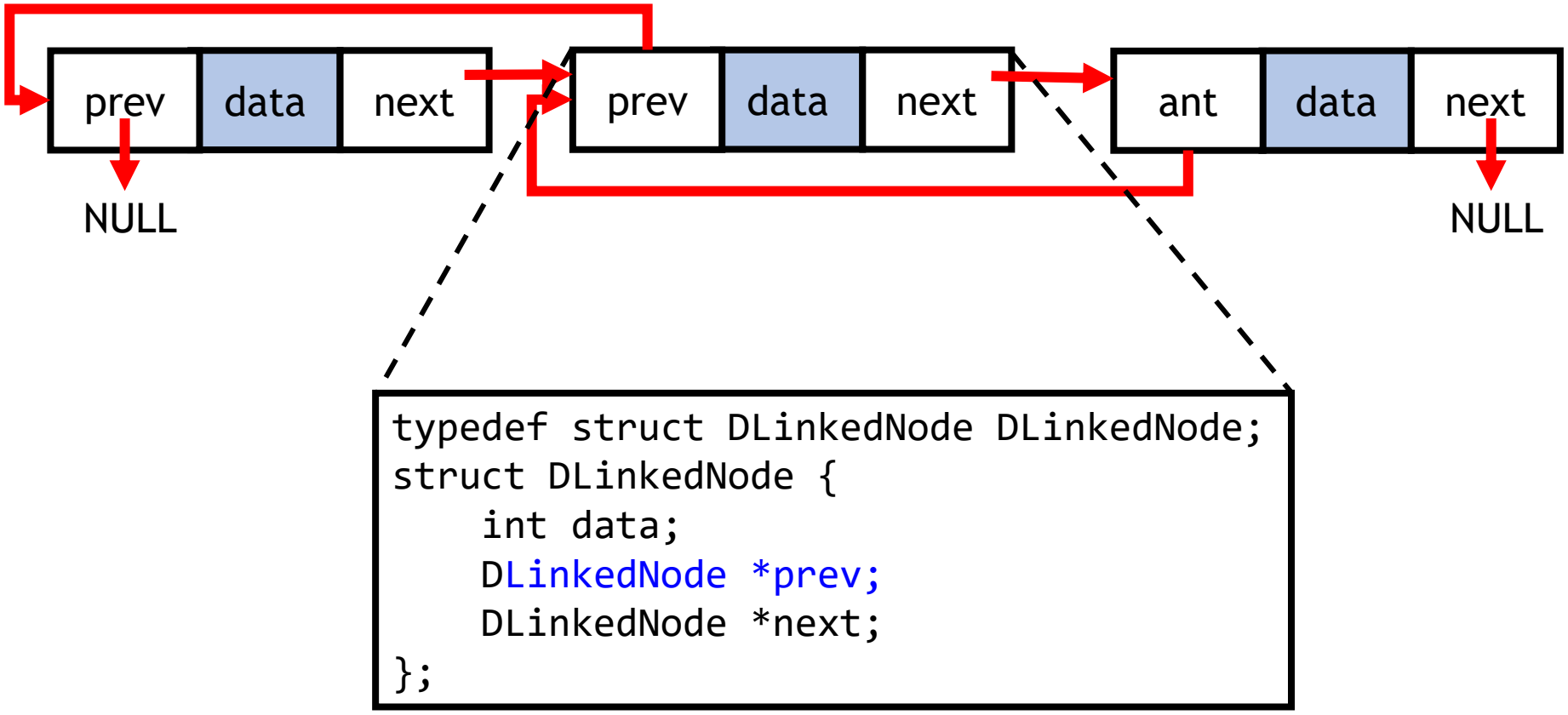
- Cada item é ligado ao próximo item e também ao anterior;



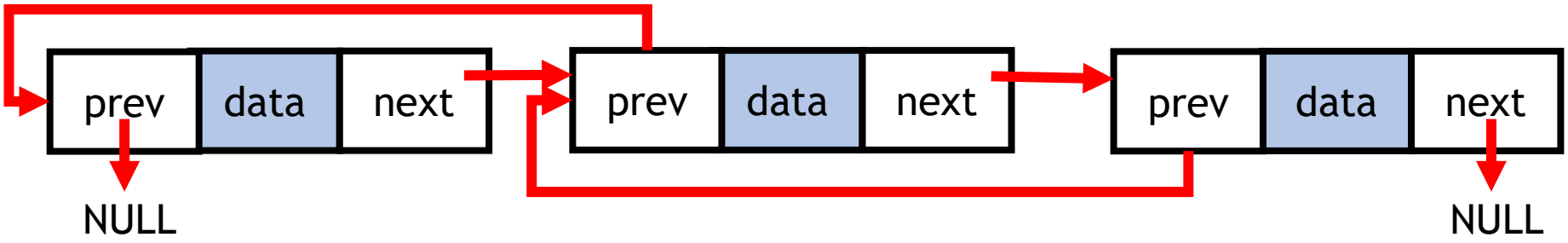
Como implementar no C?

Listas duplamente ligadas

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



Listas duplamente ligadas

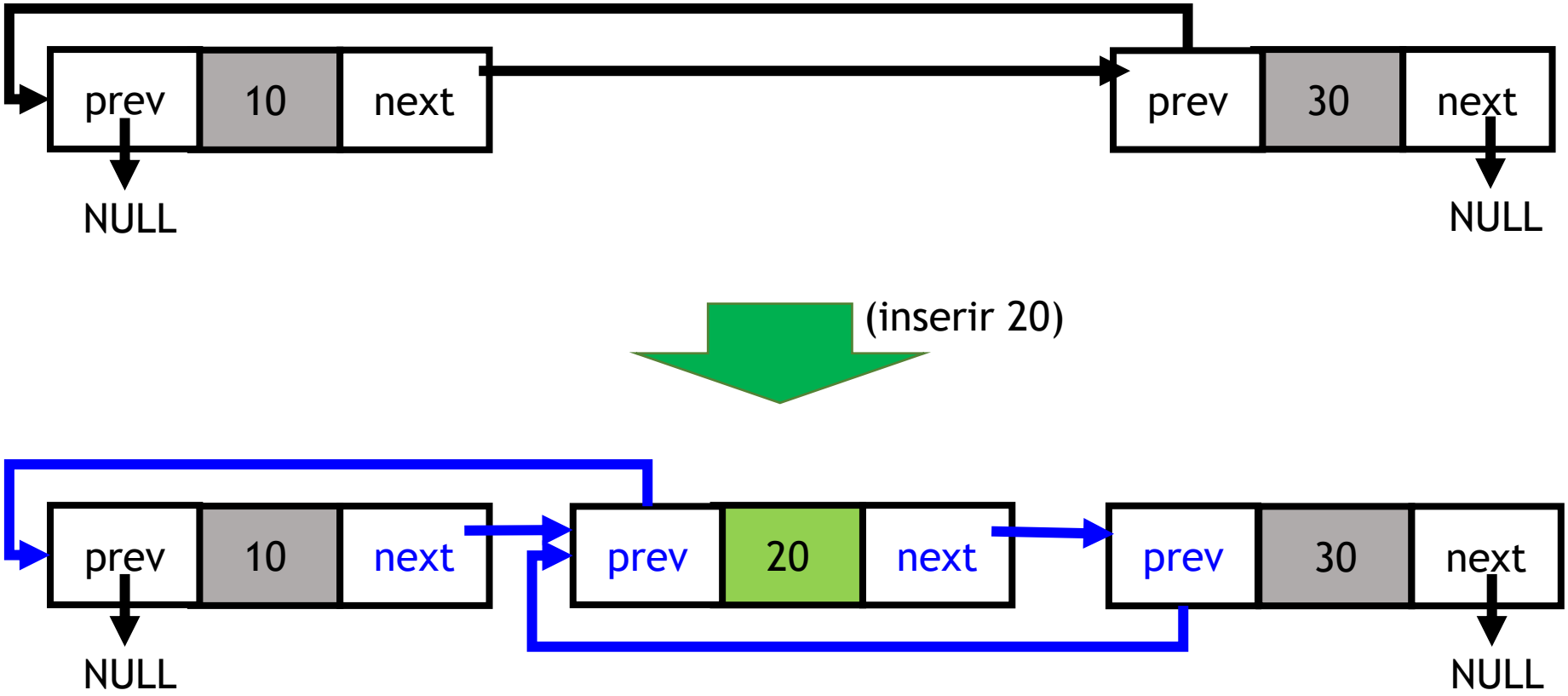



```
typedef struct DLinkedNode DLinkedNode;
struct DLinkedNode {
    int data;
    DLinkedNode *prev;
    DLinkedNode *next;
};
```

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

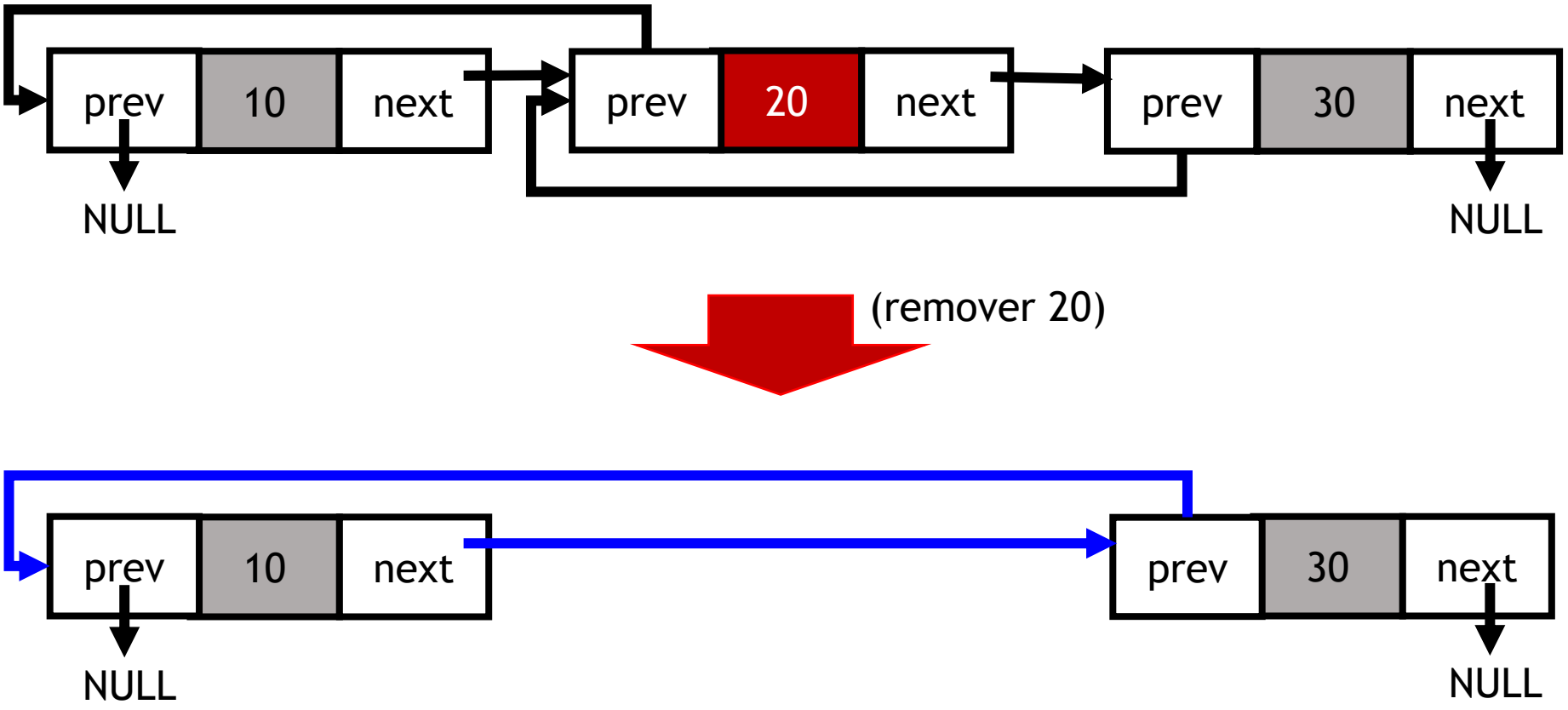
Listas duplamente ligadas

- 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);
```

```
DLinkedNode *inserir_final(DLinkedNode *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);
```

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

    if (inicio == NULL) return novo;

    DLinkedNode *anterior = NULL;
    DLinkedNode *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);
```

```
DLinkedNode *remover(DLinkedNode *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);
```

```
DLinkedNode *remover(DLinkedNode *inicio, int valor) {
    DLinkedNode *atual = inicio;
    while (atual != NULL && atual->data != valor)
        atual = atual->next;
    if (atual != NULL) {
        DLinkedNode *anterior = atual->prev;
        DLinkedNode *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 {
    DLinkedNode *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);
```

```
DLinkedNode *inserir_final_r(DLinkedNode *inicio, int valor) {
    if (inicio == NULL) {
        DLinkedNode *novo = malloc(sizeof(DLinkedNode));
        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);
```

```
DLinkedNode *remover_r(DLinkedNode *inicio, int valor) {
    if (inicio == NULL) return NULL;
    if (inicio->data == valor) {
        DLinkedNode *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.

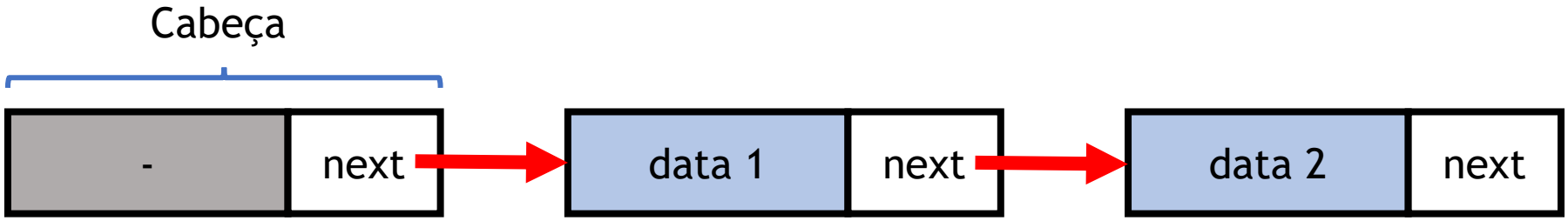
```
typedef int TIPO;

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**

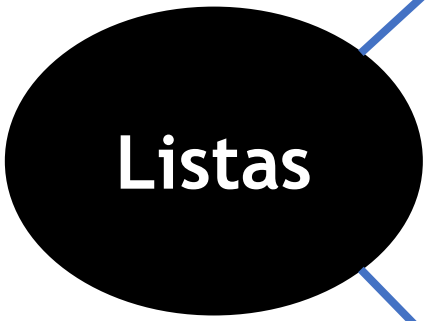
- Cada item é ligado somente ao próximo item;
- O primeiro item não armazena itens da lista (e nunca é excluído);
- **Vantagem**: não é necessário verificar se a lista está vazia (o item cabeça nunca é removido).



Listas circulares

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

Resumo



Listas com arranjos

- Simples para usar
- Alocação em bloco contínuo
- Acesso a um item em tempo constante

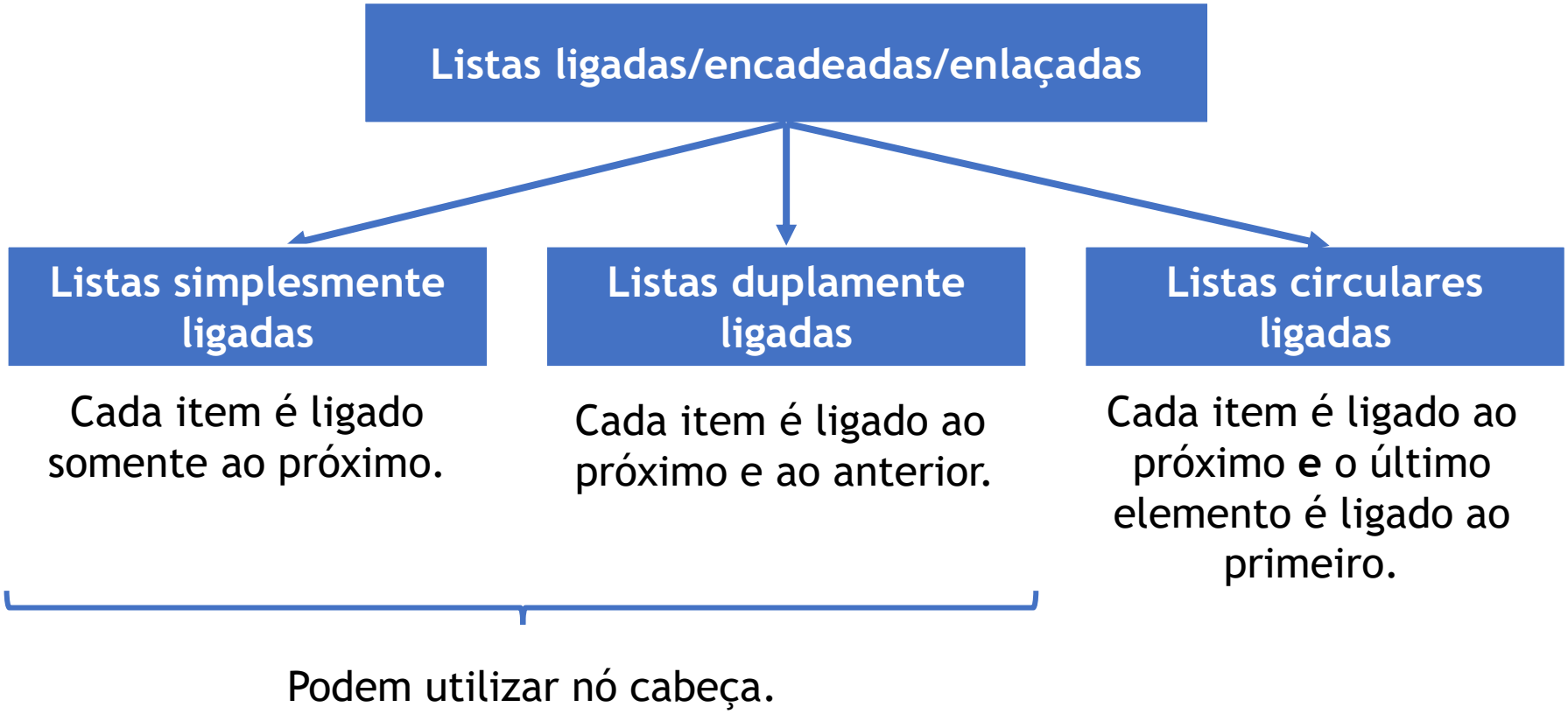
- Requer saber a quantidade de itens previamente (para alocaçao)
- Inserção/Remoção requer deslocamentos
- Expansão custosa (realocar e copiar)

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


Listas ligadas/encadeadas/enlaçadas
Listas simplesmente ligadas
Listas duplamente ligadas
Listas circulares ligadas
Cada item é ligado somente ao próximo.
Cada item é ligado ao próximo e ao anterior.
Cada item é ligado ao próximo **e** o último elemento é ligado ao primeiro.
Podem utilizar nó cabeça.

Referências

- Slides de Algoritmos e Estruturas de Dados I (UFABC): Paulo H. Pisani, Mirtha L. Venero. 2018.
- 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.