Listas ligadas (parte 1)

Prof. Paulo Henrique Pisani

novembro/2021

Tópicos

- Listas com vetor/arranjo;
- Listas ligadas (parte 1):
  - Listas ligadas;
  - Algumas operações em listas ligadas.

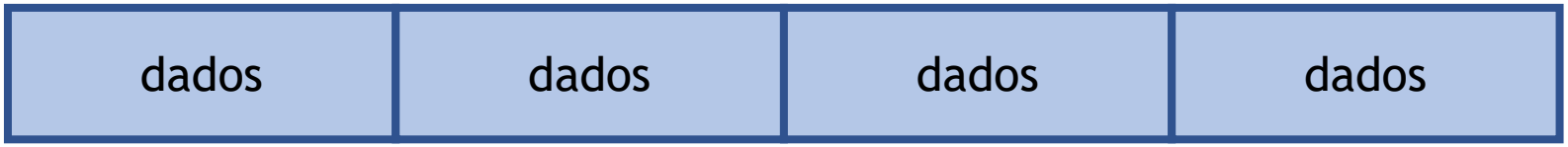
Lista

- 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

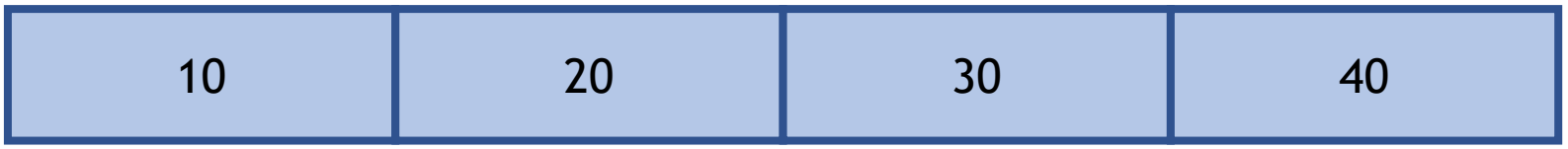
Listas com vetor/arranjo

- Itens dispostos em um arranjo sequencial;



Busca (lista com vetor)

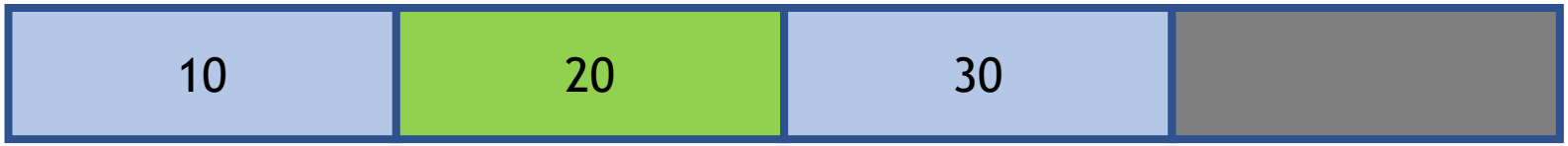
- Por exemplo, buscar o item 30 na lista. Podemos realizar **busca sequencial**;
- É possível acessar elementos diretamente (com índice), o que pode permitir **busca binária (vetor ordenado)**.



Inserir (lista com vetor)

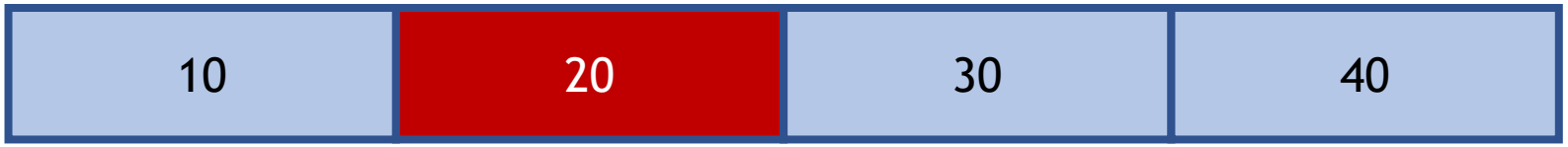
- Inserir um item pode precisar de deslocamentos;

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

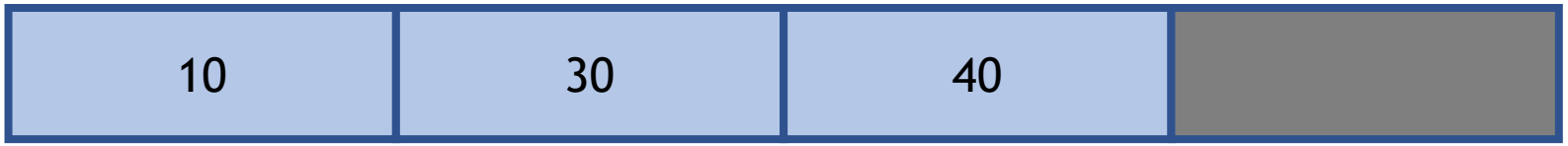


Remover (lista com vetor)

- Remover um item pode precisar de deslocamentos;

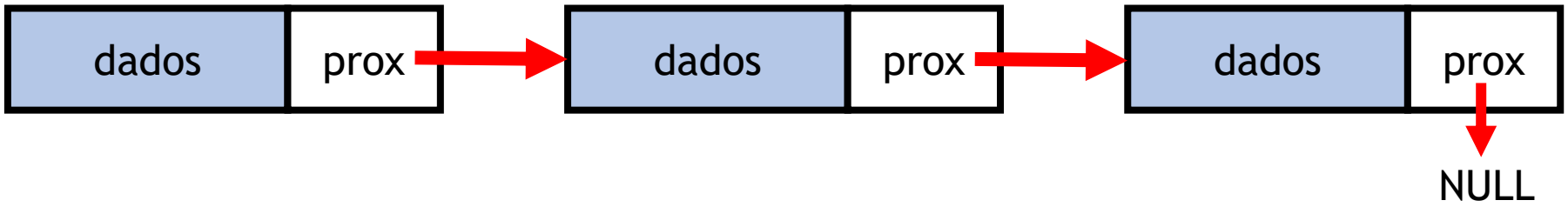


Remover item 20, envolveu o deslocamento do 30 e do 40.



Listas ligadas/encadeadas

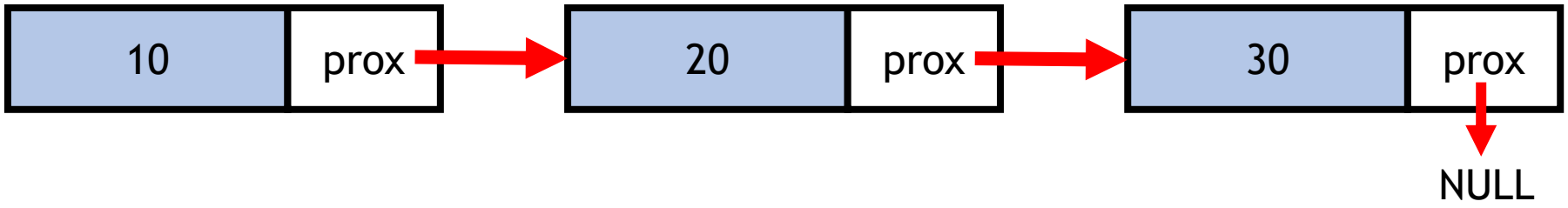
- Estrutura de dados que armazena os itens de forma não consecutiva na memória:
  - Cada item possui uma referência para o próximo.



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.

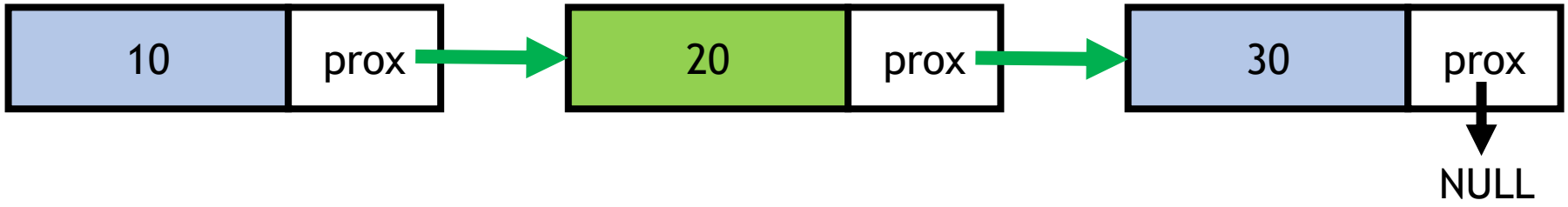


Por exemplo, acessar o terceiro item: Com a referência do início da lista, é necessário ir até o segundo para só depois acessar o terceiro.

Inserir (lista ligada)

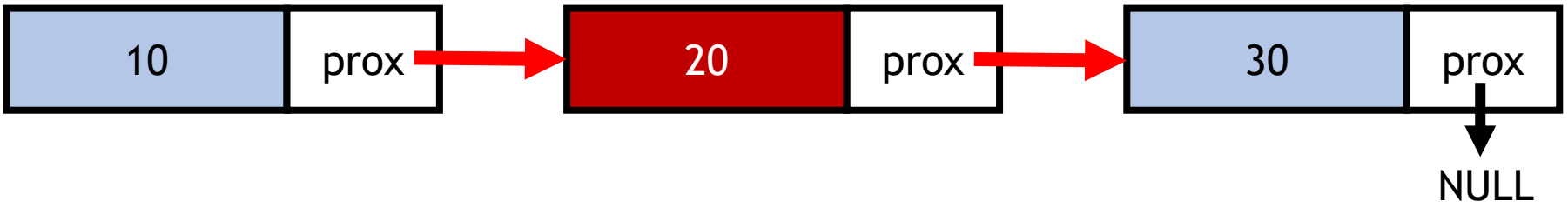
- Não requer deslocamentos para inserção e remoção.

Inserir item 20



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.

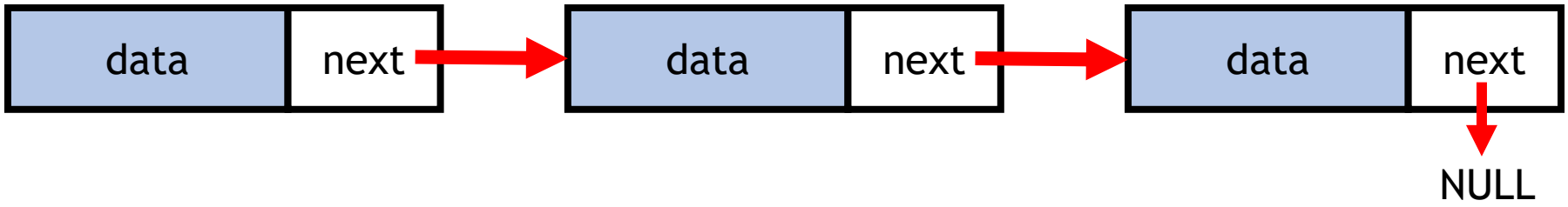
Listas ligadas/encadeadas

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

Listas ligadas

Listas ligadas

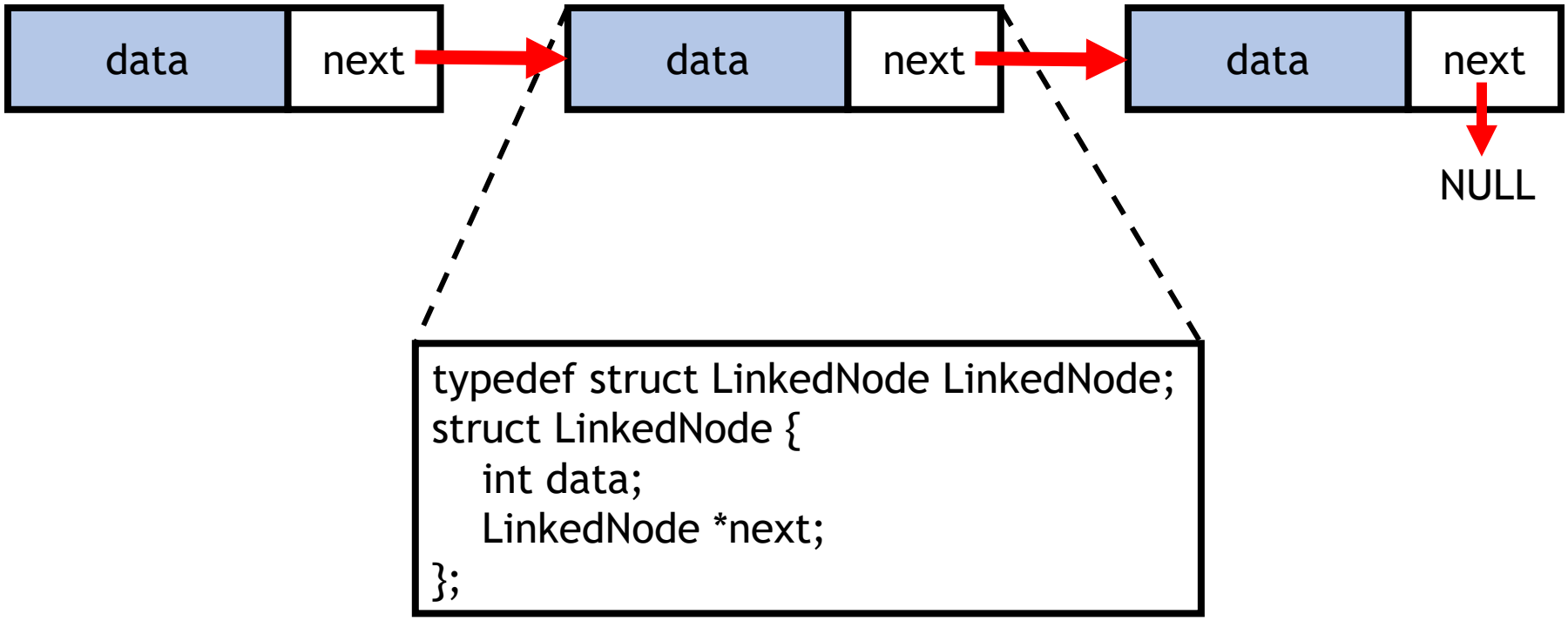
- Cada item é ligado ao próximo item.



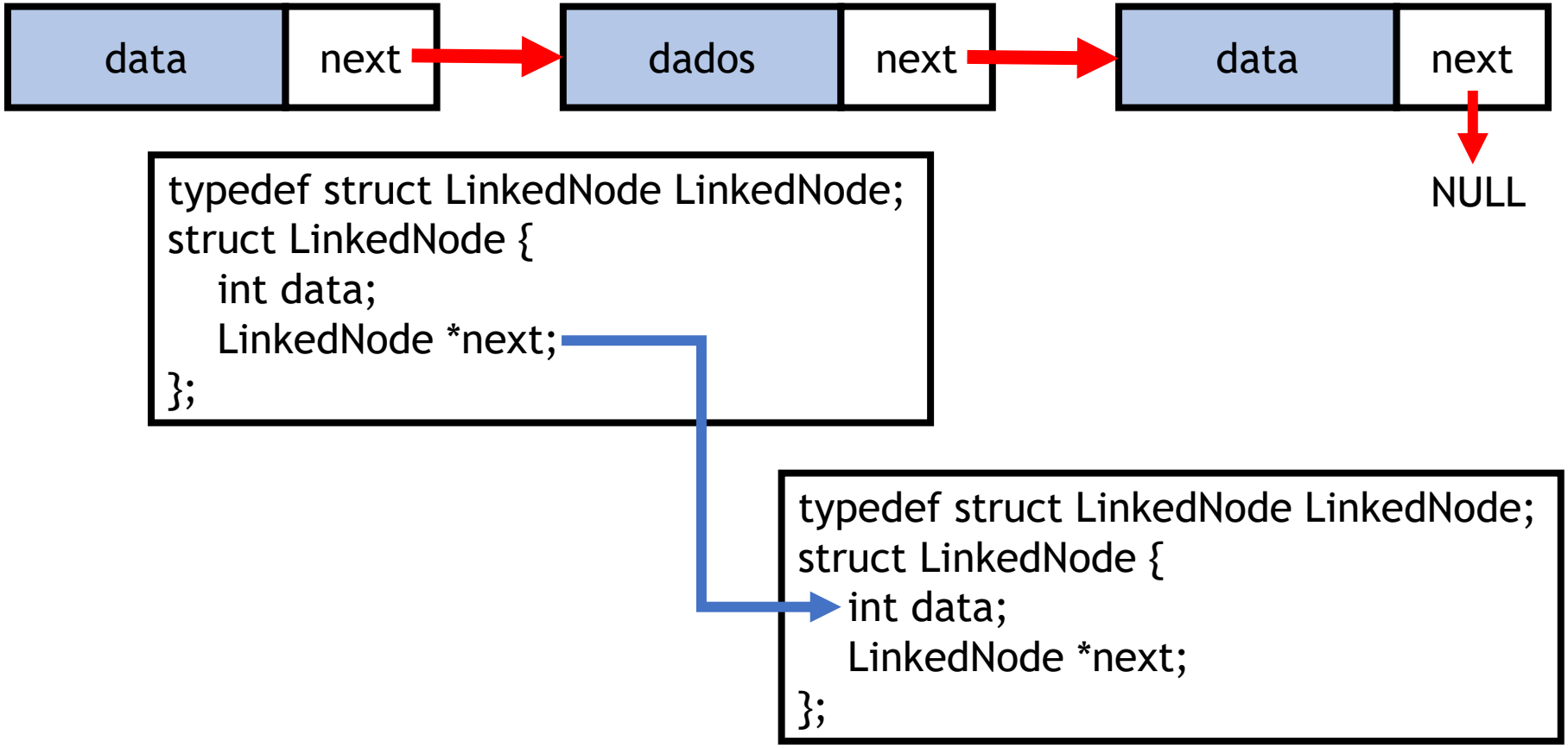
Como implementar no C?

Listas ligadas

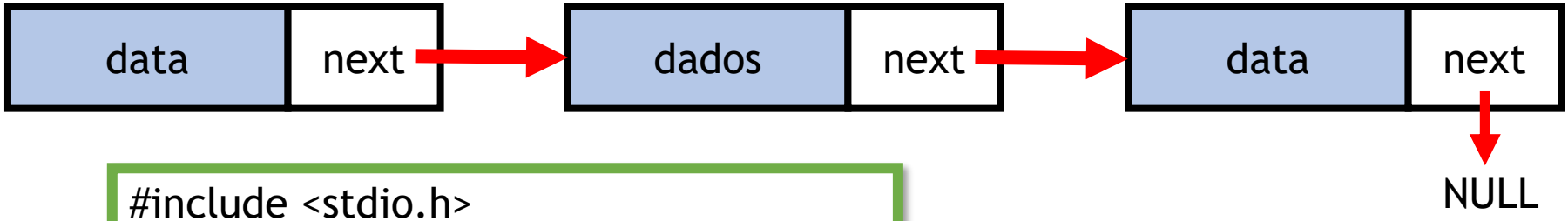
- Cada item é ligado ao próximo item.



Listas ligadas



Listas ligadas




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

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

int main() {
      LinkedNode *inicio = NULL;

      …
}
```

O ponteiro para o primeiro item deve ser salvo.

Listas ligadas


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

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

int main() {
      LinkedNode *inicio = 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);
```

```
LinkedNode *inserir_final(LinkedNode *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);
```

```
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);
```

```
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;
}
```

Aloca novo nó

Se a lista é vazia, o novo nó é o primeiro

Percorre lista, **anterior** terá o ponteiro para último nó

next do último nó recebe o novo nó

Inserir no final

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

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

Veremos agora uma versão recursiva

```
}
```

Inserir no final

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

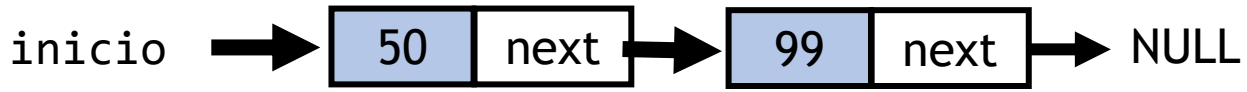
```
LinkedNode *inserir_final_r(LinkedNode *inicio, int valor) {
    if (inicio == NULL) {
        LinkedNode *novo = malloc(sizeof(LinkedNode));
        if (novo == NULL) return inicio;
        novo->data = valor;
        novo->next = NULL;
        return novo;
    }
    inicio->next = inserir_final_r(inicio->next,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

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

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

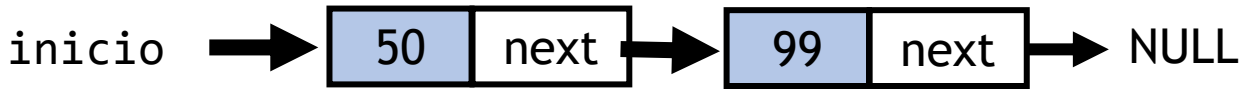
O inicio não é alterado, pois a lista não é vazia.

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




```
LinkedNode *inserir_final_r(Li 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(Li
      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;
}
```

50 next

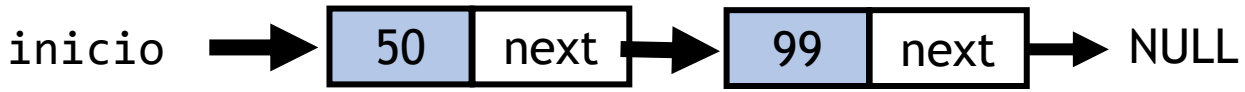
int valor) {

```
LinkedNode *inserir_final_r(Li
      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;
}
```

99 next

int valor) {

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



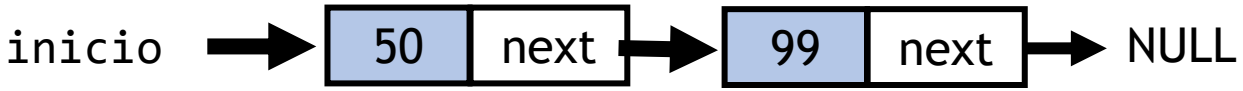

```
LinkedNode *inserir_final_r(LinkedNode *inicio, int valor) {
      if (inicio == NULL) {
            LinkedNode *novo = malloc(sizeof(LinkedNode));
            if (novo == NULL) return inicio;
            novo->data = valor;
            novo->next = NULL;
            return novo;
      }
      inicio->next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

50 next

99 next

NULL

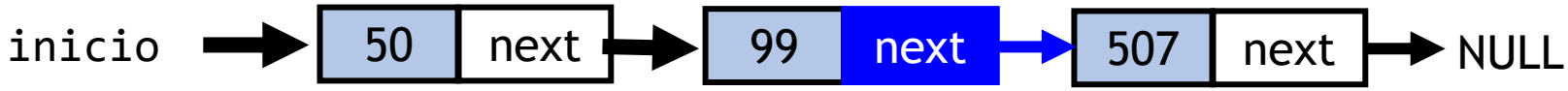
Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.




```
LinkedNode *inserir_final_r(Li 50 next 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 = LinkedNode *inserir_final_r(Li 99 next 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 = LinkedNode *inserir_final_r(L NULL int valor) {
                                          if (inicio == NULL) {
                                                LinkedNode *novo = malloc(sizeof(LinkedNode));
                                                if (novo == NULL) return inicio;
                                                novo->data = valor;
                                                novo->next = NULL;
                                                return 507 next
                                          }
                                          inicio->next = inserir_final_r(inicio->next, valor);
                                          return inicio;
                                    }
                        return inicio;
                  }
      return inicio;
}
```

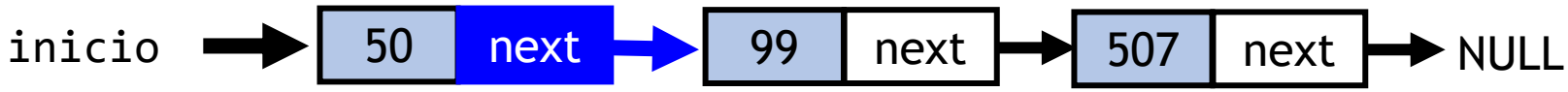
Aloca novo nó

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




```
LinkedNode *inserir_final_r(Li 50 next  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 = LinkedNode *inserir_final_r(Li 99 next  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 = 507 next   next de 99 aponta para o 507

                        return inicio;
                  }
      return inicio;
}
```

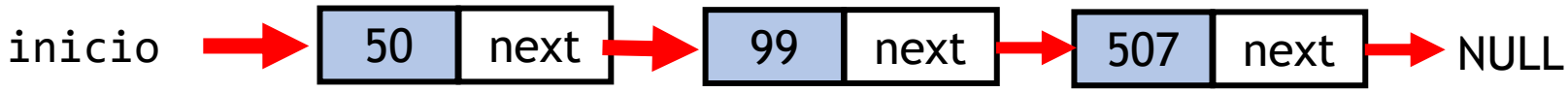
Veremos o efeito nesta lista exemplo. Vamos adicionar o 507.




```
LinkedNode *inserir_final_r(Li 50 next  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 = 99 next   next de 50 aponta para o 99

      return inicio;
}
```

(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);
```

```
LinkedNode *remover(LinkedNode *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);
```

Procura nó

Ajusta ponteiros

Libera nó

```
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;
}
```

Remover

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

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

Veremos agora uma versão recursiva

```
}
```

Remover

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

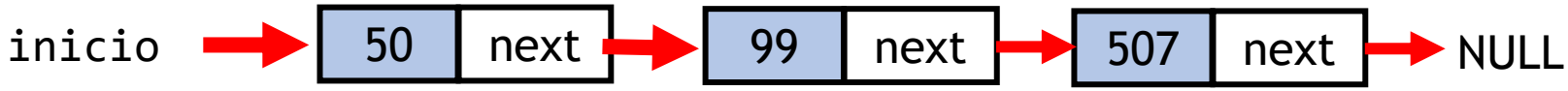
```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {
    if (inicio == NULL) return NULL;
    if (inicio->data == valor) {
        LinkedNode *temp = inicio->next;
        free(inicio);
        return temp;
    }
    inicio->next = remover_r(inicio->next, valor);
    return inicio;
}
```

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

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

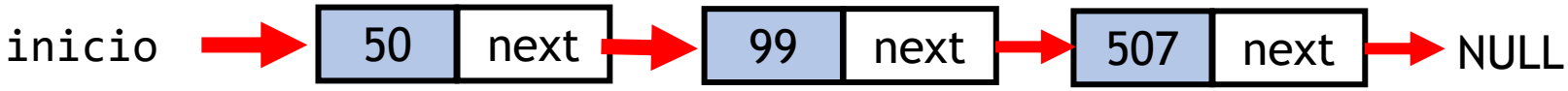
O inicio não é alterado.

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




```
LinkedNode *remover_r(Li                    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.



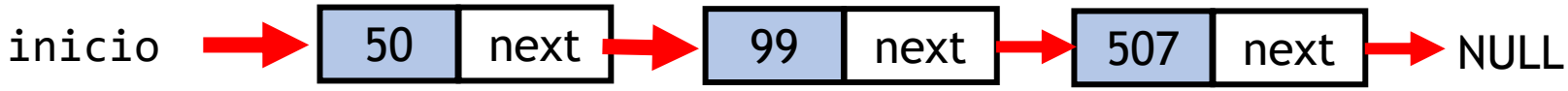

```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {
      if (inicio == NULL) return NULL;
      if (inicio->data == valor) {
            LinkedNode *temp = inicio->next;
            free(inicio);
            return temp;
      }
      inicio->next = remover_r(inicio->next, valor);

      return inicio;
}
```

50 next

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;
}
```

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

99 next

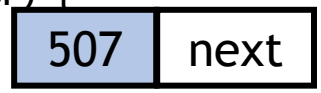
```
LinkedNode *remover_r(LinkedNode *inicio, int valor) {
      if (inicio == NULL) return NULL;
      if (inicio->data == valor) {
            LinkedNode *temp =
            free(inicio);
            return temp;
      }
      inicio->next = remover_r(inicio->next, valor);
      return inicio;
}
```

507 next

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


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

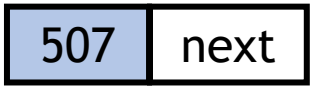



```
      return inicio;
}
```

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


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

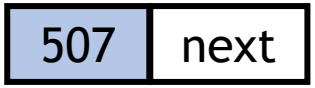

```
LinkedNode *remover_r(Li                    int valor) {
      if (inicio == NULL) return NULL;
      if (inicio->data == valor) {
            LinkedNode *temp = inicio->next;
            free(inicio);
            return temp;
      }
      inicio->next = remover_r(inicio->next, valor);
      return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




```
      return inicio;
}
```

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


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



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.

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.