Listas ligadas

Prof. Paulo Henrique Pisani

http://professor.ufabc.edu.br/~paulo.pisani/

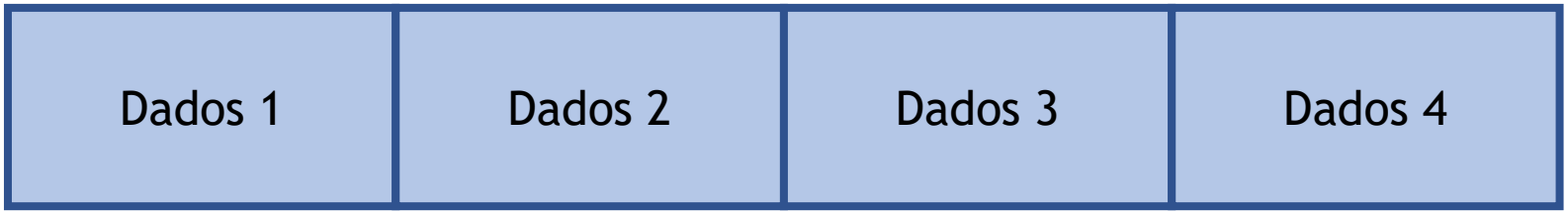
fevereiro/2019

Agenda

- Listas com arranjos;
- Listas ligadas/encadeadas:
  - Listas simplesmente ligadas;
  - Listas duplamente ligadas;
  - Outros tipos: Listas com nó cabeça e Listas circulares.
- Exercícios.

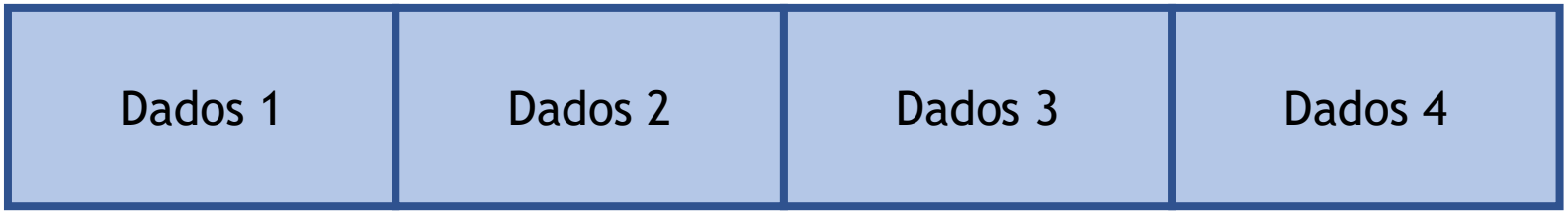
Listas com arranjos (revisão)

- Itens dispostos em um arranjo sequencial;



Listas com arranjos (revisão)

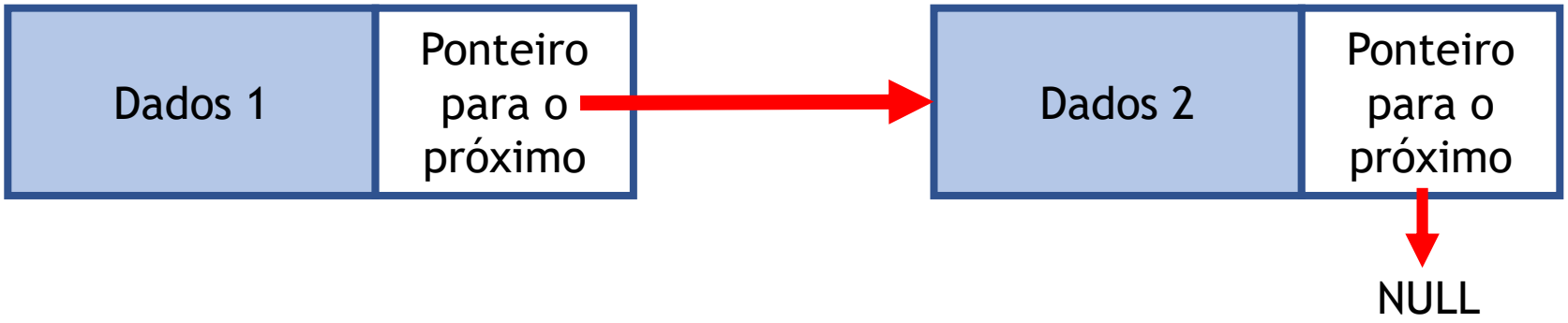
- Itens dispostos em um arranjo sequencial;



Problemas?

Listas ligadas/encadeadas

- Estrutura de dados que armazena os itens de forma não consecutiva na memória:
  - Usa ponteiros para “ligar” um item no próximo.



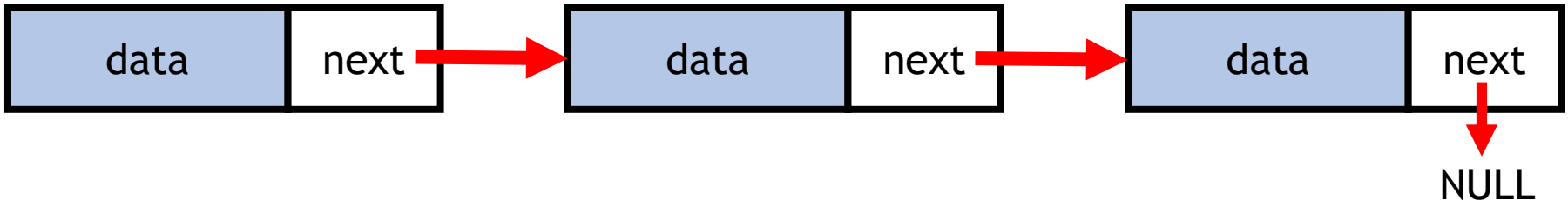
Listas ligadas/encadeadas

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

Listas simplesmente ligadas

Listas simplesmente ligadas

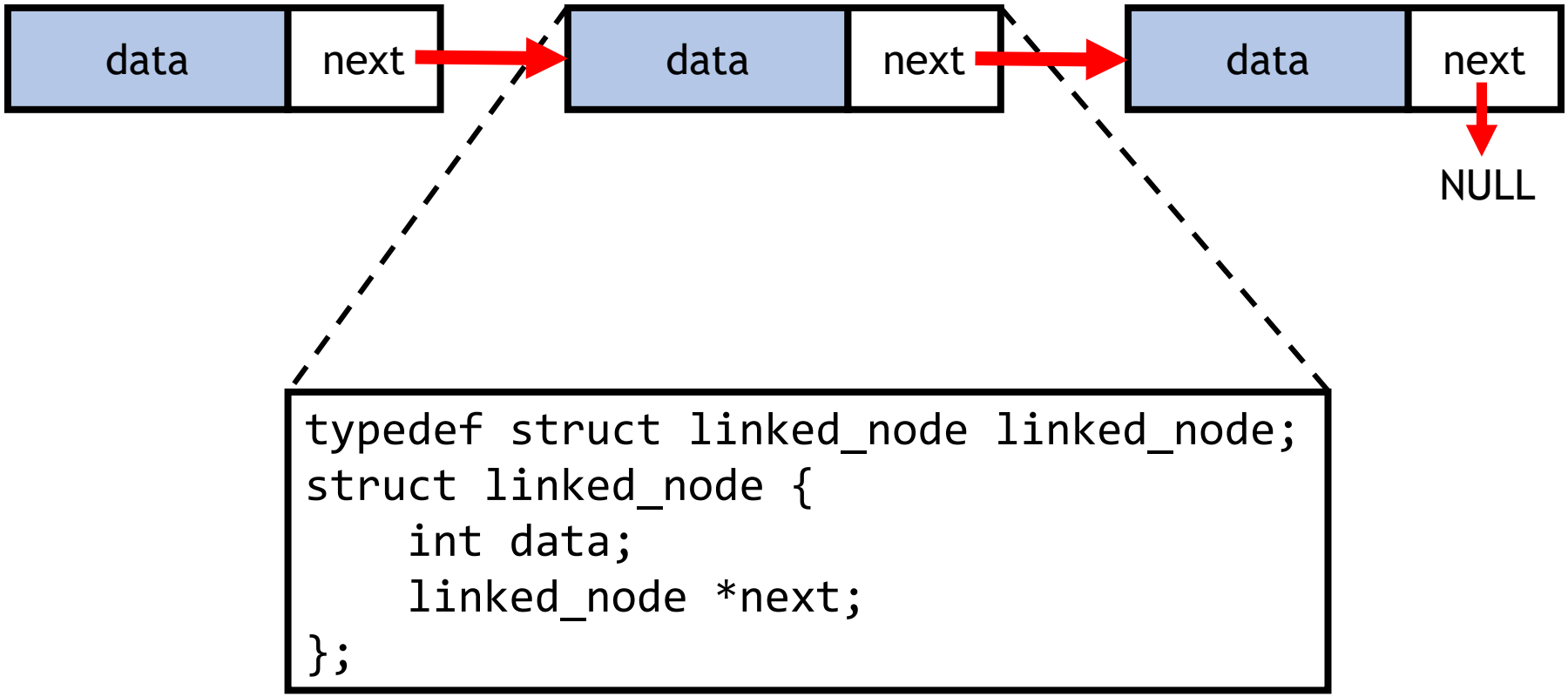
- Cada item é ligado somente ao próximo item;



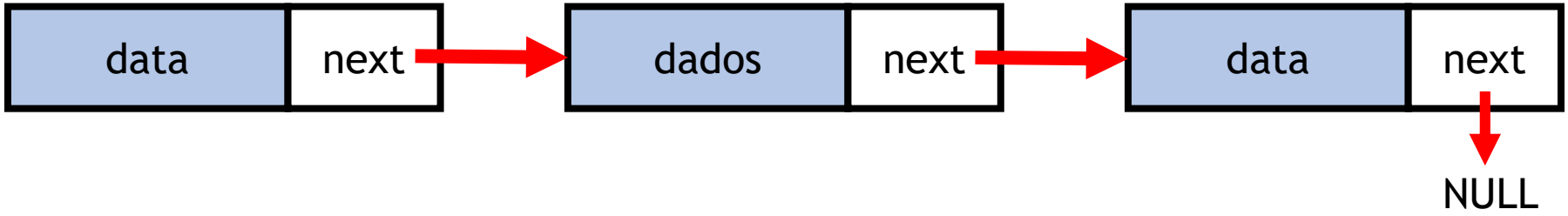
Como implementar no C?

Listas simplesmente ligadas

- Cada item é ligado somente ao próximo item;



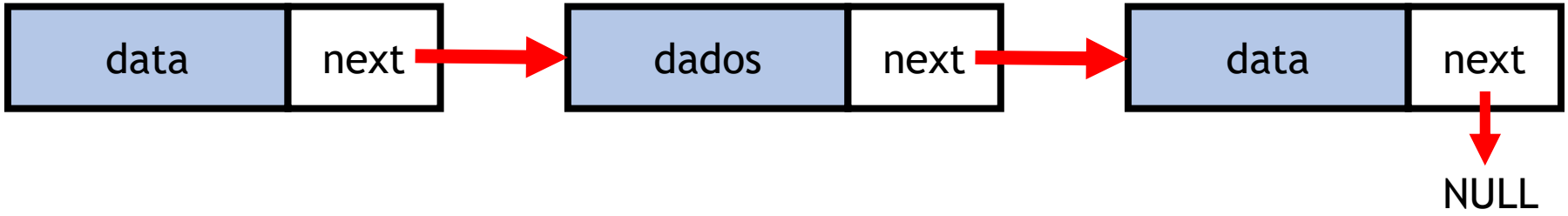
Listas simplesmente ligadas




```
struct linked_node {
    int data;
    linked_node *next;
};
```

```
struct linked_node {
    int data;
    linked_node *next;
};
```

Listas simplesmente ligadas



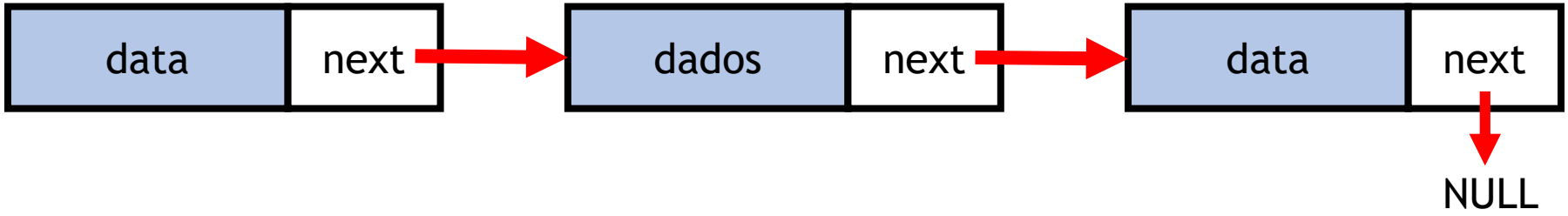
Lista vazia

```
linked_node *lista = NULL;
```

Lista com um elemento

```
linked_node *lista = malloc(sizeof(linked_node));
lista->next = NULL
```

Listas simplesmente ligadas



Lista vazia

```
linked_node *inicio = NULL;
```

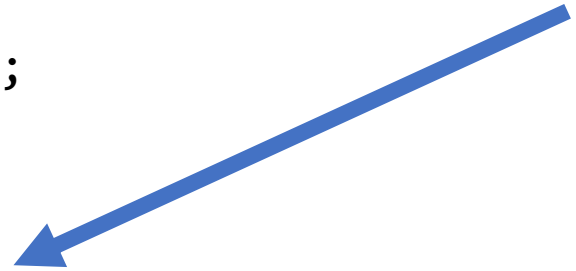
Lista com um elemento

```
linked_node *inicio = malloc(sizeof(linked_node));
inicio->next = NULL
```

Importante: Precisamos armazenar o ponteiro para o **primeiro elemento da lista ligada!** Devido a isso, chamaremos esse primeiro ponteiro de “inicio” ao invés de “lista”

Listas simplesmente ligadas

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

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

int main() {
    linked_node *inicio = malloc(sizeof(linked_node));
    inicio->data = 1;
    inicio->next = NULL;


    return 0;
}
```



O ponteiro para o primeiro item deve ser salvo.

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

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

int main() {
    linked_node *inicio = malloc(sizeof(linked_node));
    inicio->data = 1;
    inicio->next = NULL;

    linked_node *segundo = malloc(sizeof(linked_node));
    segundo->data = 2;
    segundo->next = NULL;

    inicio->next = segundo;

    return 0;
}
```

Cria segundo elemento

Liga o primeiro ao segundo

Imprimir lista

- Começamos no primeiro elemento, depois imprimimos o próximo e assim por diante; até que chegaremos ao último elemento.
- **Como saber qual é o último elemento de uma lista simplesmente ligada?**

Imprimir lista

- Começamos no primeiro elemento, depois imprimimos o próximo e assim por diante; até que chegaremos ao último elemento.
- **Como saber qual é o último elemento de uma lista simplesmente ligada?**

É o elemento com **`next = NULL`**

Imprimir lista

- Começamos no primeiro elemento, depois imprimimos o próximo e assim por diante; até que chegaremos ao último elemento.

```
void imprimir_lista(linked_node *inicio) {
   linked_node *atual = inicio;
   while (atual != NULL) {
      printf("%d ", atual->data);
      atual = atual->next;
   }
   printf("\n");
}
```

Imprimir lista

- Começamos no primeiro elemento, depois imprimimos o próximo e assim por diante; até que chegaremos ao último elemento.

```
void imprimir_lista(linked_node *inicio) {
   linked_node *atual = inicio;
   while (atual != NULL) {
      printf("%d ", atual->data);
      atual = atual->next;
   }
   printf("\n");
}
```

E como ficaria uma versão recursiva dessa função?

Imprimir lista

- Começamos no primeiro elemento, depois imprimimos o próximo e assim por diante; até que chegaremos ao último elemento.

```
void imprimir_lista_rec(linked_node *atual) {
   if (atual == NULL) {
      printf("\n");
   } else {
      printf("%d ", atual->data);
      imprimir_lista_rec(atual->next);
   }
}
```

E como ficaria uma versão recursiva dessa função?

Função para adicionar ao final

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

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

    if (inicio == NULL)  return novo;

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

    anterior->next = novo;
    return inicio;
}
```

Função para adicionar ao final

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

```
linked_node *append_node(linked_node *inicio, int valor) {
    linked_node *novo = malloc(sizeof(linked_node));
    novo->data = valor;
    novo->next = NULL;

    if (inicio == NULL) {
        return novo;
    }

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

    anterior->next = novo;
    return inicio;
}
```

```
// Uso da funcao append_node

linked_node *inicio = NULL;
inicio = append_node(inicio, 1);
inicio = append_node(inicio, 2);
inicio = append_node(inicio, 3);
```

Função para adicionar ao final (2)

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

```
linked_node *append_node_l(linked_node *ultimo, int valor) {
    linked_node *novo = malloc(sizeof(linked_node));
    if (novo == NULL) return NULL;
    novo->data = valor;
    novo->next = NULL;

    if (ultimo != NULL) ultimo->next = novo;

    return novo;
}
```

Função para adicionar ao final (2)

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

```
// Uso da funcao append_node_l

linked_node *inicio = NULL, *fim = NULL;
fim = append_node_l(fim, 1);
inicio = fim;  // Inicializa o "inicio"
fim = append_node_l(fim, 2);
fim = append_node_l(fim, 3);
```

Exercício 1 – adicionar no final

- Uma sequência de ADN é formada por quatro bases (A, C, G, T). Escreva um programa que leia uma sequência de bases até que o usuário digite "0". A sequência deverá ser armazenada em memória. Depois imprima a sequência.

Use uma lista simplesmente ligada

Liberar lista

- Como fazemos para liberar a lista da memória?

Liberar lista

- Como fazemos para liberar a lista da memória?

```
void liberar_lista(linked_node *atual) {
   if (atual != NULL) {
      liberar_lista(atual->next);
      free(atual);
   }
}

// Chamada
liberar_lista(inicio);
```

Exercício 2 – adicionar ordenado

- Modifique o exercício anterior de modo que a lista ligada armazene as bases em ordem alfabética. Portanto, se o usuário digitar A, G, T, A, C, T, C, G, T, o programa armazenará A, A, C, C, G, G, T, T, T na lista ligada.

Use apenas uma lista simplesmente ligada

Exemplo

Entrada do usuário	AGTACTCGT0
Saída do programa	AACCGGTTT

Exercício 3 – remover

- Um disco voador abandonado foi encontrado pelo professor ABC. Dentre do disco, havia vestígios do ADN alienígena, que também era formado pelas quatro bases A, C, G, T.
- Contudo, quando entrou em contato com a água, as bases C e G foram eliminadas da sequência.
- Escreva um programa que leia a sequência de ADN alienígena e simule o processo de eliminação das bases C e G. Depois imprima a sequência antes e depois do processo.

Use APENAS UMA lista simplesmente ligada

Exercício 4 – estrutura do nó

- Apesar da vantagem de ter tamanho indefinido, a lista ligada nos exercícios anterior implica em um custo elevado de memória. **Por-que?**

Exercício 4 – estrutura do nó

- Apesar da vantagem de ter tamanho indefinido, a lista ligada nos exercícios anterior implica em um custo elevado de memória. **Por-que?**

Cada elemento armazena **um char + um ponteiro (next)**, ou seja:
32-bit: 4 bytes (1 byte) + 4 bytes = 8 bytes
64-bit: 8 bytes (1 byte) + 8 bytes = 16 bytes

Portanto, cada elemento consome 8 (32-bit) ou 16 bytes (64-bit)!

Exercício 4 – estrutura do nó

- Apesar da vantagem de ter tamanho indefinido, a lista ligada nos exercícios anterior implica em um custo elevado de memória. **Por-que?**

Cada elemento armazena **um char + um ponteiro (next)**, ou seja:
32-bit: 4 bytes (1 byte) + 4 bytes = 8 bytes
64-bit: 8 bytes (1 byte) + 8 bytes = 16 bytes

Como podemos melhorar a eficiência de uso de memória nesse problema?

Exercício 4 – estrutura do nó

Exercício de Profa. Mirtha Lina Fernández Venero

- Reimplemente os exercícios anteriores utilizando a estrutura **dna_linked_node** para o nó:

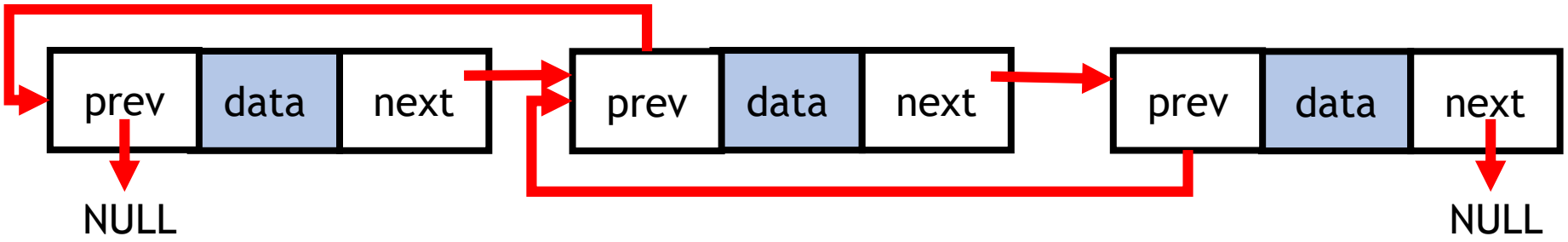
```
typedef struct dna_data dna_data;
struct dna_data {
    char bases[16];
    int stored_items;
};

typedef struct dna_linked_node dna_linked_node;
struct dna_linked_node {
    dna_data data;
    dna_linked_node *next;
};
```

Quantidade de bases armazenadas

Listas duplamente ligadas

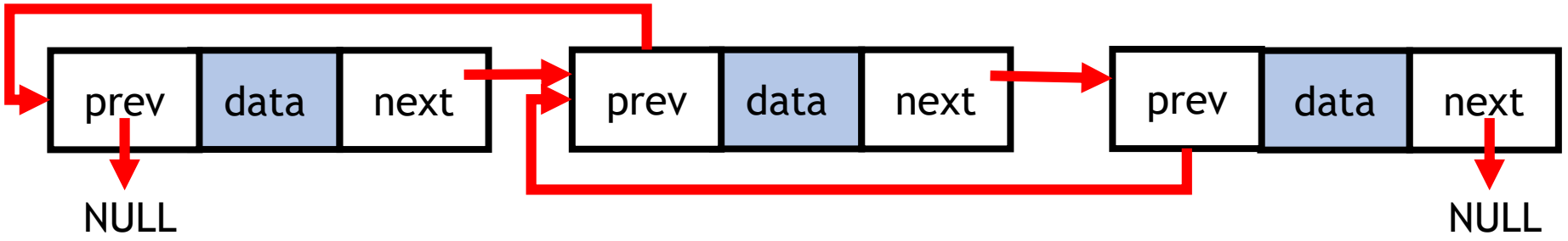
Listas duplamente ligadas

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



Listas duplamente ligadas

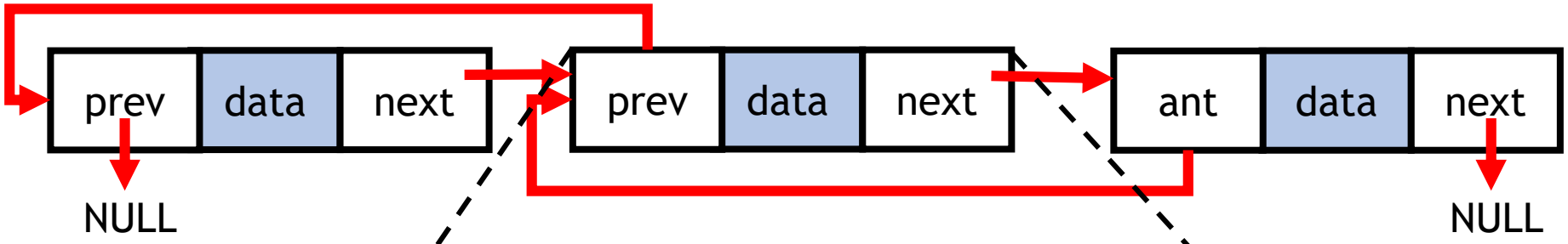
- Cada item é ligado ao próximo item e também ao anterior;



Como implementar no C?

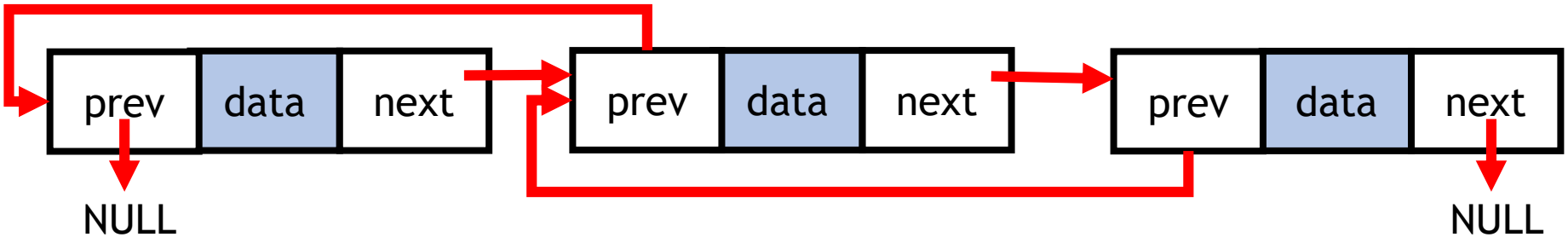
Listas duplamente ligadas

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




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

Listas duplamente ligadas




```
struct d_linked_node {
    int data;
    d_linked_node *prev;
    d_linked_node *next;
};
```

```
struct d_linked_node {
    int data;
    d_linked_node *prev;
    d_linked_node *next;
};
```

Função para adicionar ao final

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

    if (inicio == NULL) return novo;

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

    anterior->next = novo;
    novo->prev = anterior;

    return inicio;
}
```

Função para adicionar ao final

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

    anterior->next = novo;
    novo->prev = anterior;

    return inicio;
}
```

```
// Uso da funcao append_node_d

d_linked_node *inicio = NULL;
inicio = append_node_d(inicio, 1);
inicio = append_node_d(inicio, 2);
inicio = append_node_d(inicio, 3);
```

Função para adicionar ao final (2)

```
d_linked_node *append_node_l_d(d_linked_node *ultimo, int valor) {
    d_linked_node *novo = malloc(sizeof(d_linked_node));
    if (novo == NULL) return NULL;
    novo->data = valor;
    novo->prev = NULL;
    novo->next = NULL;

    if (ultimo != NULL) {
        ultimo->next = novo;
        novo->prev = ultimo;
    }

    return novo;
}
```

Função para adicionar ao final (2)

```
d_linked_node *append_node_l_d(d_linked_node *ultimo, int valor) {
    d_linked_node *novo = malloc(sizeof(d_linked_node));
    if (novo == NULL) return NULL;
```

```
// Uso da funcao append_node_l_d

d_linked_node *inicio, *fim = NULL;
fim = append_node_l_d(fim, 1);
inicio = fim;  // Inicializa o "inicio"
fim = append_node_l_d(fim, 2);
fim = append_node_l_d(fim, 3);
```

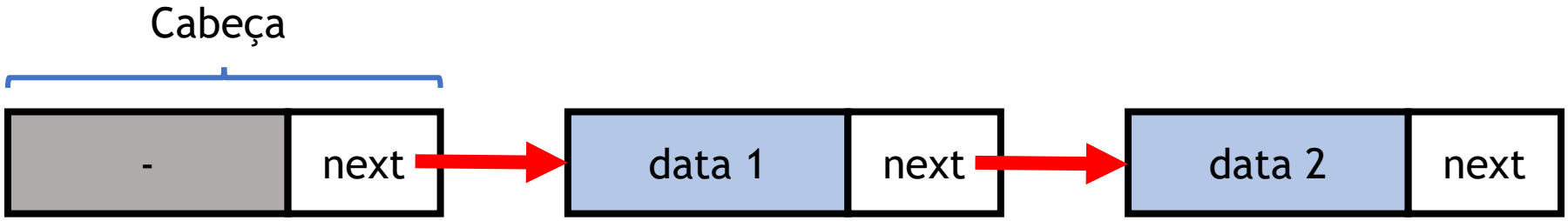
```
}
```

Outros tipos

Com nó cabeça e lista circular

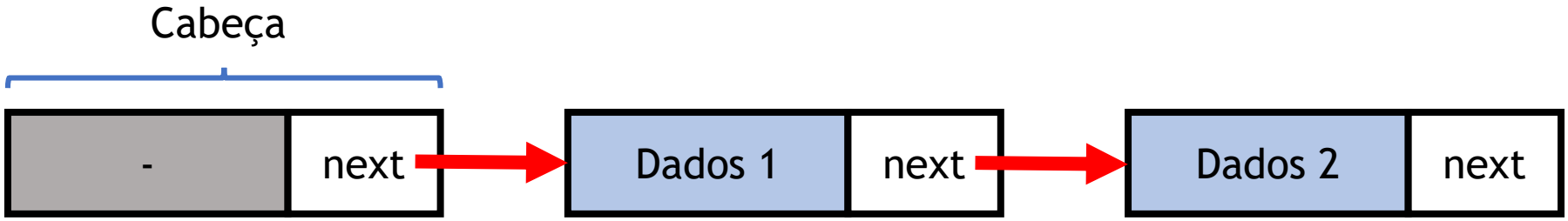
Listas simplesmente ligadas **com nó cabeça**

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



Listas simplesmente ligadas **com nó cabeça**

- Cada item é ligado somente ao próximo item;
- O primeiro item não armazena dados da lista (e nunca é excluído).



Como implementar no C?

Listas simplesmente ligadas

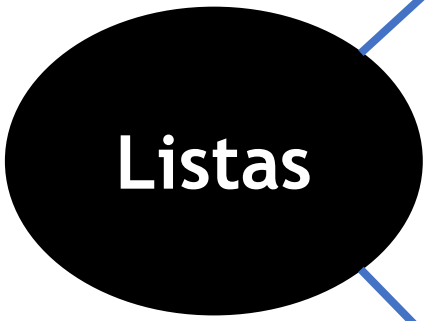
```
typedef struct linked_node linked_node;
struct linked_node {
    int data;
    linked_node *next;
};
```

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

    if (inicio == NULL) return novo;

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

    anterior->next = novo;
    return inicio;
}
```

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

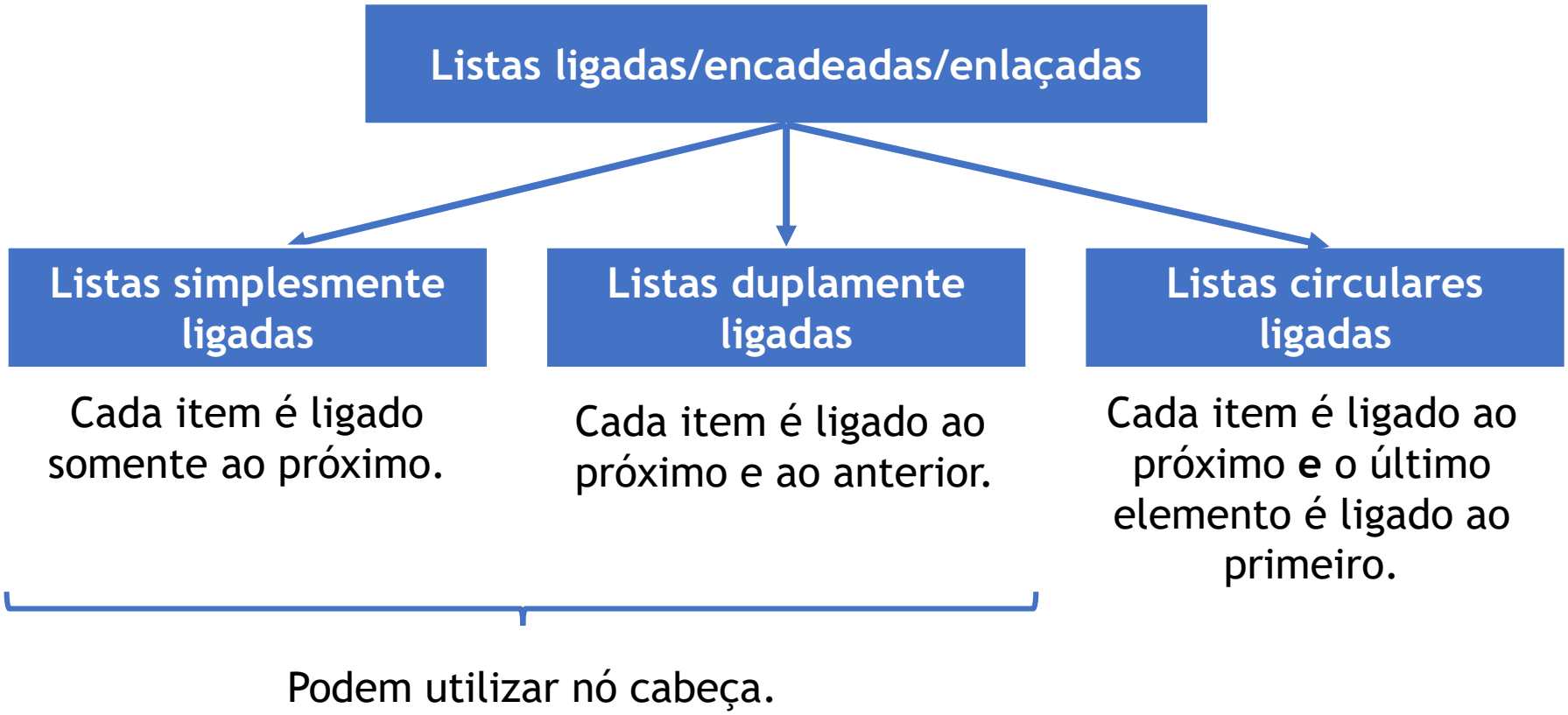
- Requer saber a quantidade de itens previamente (para alocaçao)
- Inserção/Remoção requer deslocamentos
- Expansão custosa (realocar e copiar)

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.

Exercícios

- Escreva funções em C para realizar as seguintes operações com **listas simplesmente 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.

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.
- Robert Sedgewick. Algorithms in C/C++/Java, Parts 1-4 (Fundamental Algorithms, Data Structures, Sorting, Searching). Addison-Wesley Professional.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. Elsevier, 2012.