Algoritmos e Complexidade

Aula 02: Estruturas de Dados - Homogêneas, Heterogêneas e Ponteiros

Prof. Vagner Cordeiro
Sistemas de Informação
Universidade - 2024

Agenda da Aula

1. **Estruturas de Dados: Conceitos Fundamentais**
2. **Arrays (Vetores) - Estruturas Homogêneas**
3. **Matrizes e Arrays Multidimensionais**
4. **Ponteiros: Conceitos e Aplicações**
5. **Estruturas Heterogêneas (Structs)**
6. **Unions e Manipulação de Memória**
7. **Operações e Algoritmos**
8. **Comparações C vs Python**
9. **Análise de Performance e Complexidade**

Objetivos de Aprendizagem

Ao final desta aula, o aluno será capaz de:

- **Distinguir** estruturas homogêneas e heterogêneas
- **Implementar** arrays e matrizes em C e Python
- **Dominar** conceitos de ponteiros e endereçamento
- **Criar** e manipular structs e unions
- **Analisar** complexidade de operações em estruturas
- **Aplicar** estruturas adequadas para problemas específicos

1. Estruturas de Dados: Definições Matemáticas

Estrutura de Dados

Uma estrutura de dados é uma organização de dados que suporta operações:

$$S = (D, O)$$

Onde:

- D = Conjunto de dados armazenados
- O = Conjunto de operações permitidas

Classificação Principal

- **Homogêneas:** Todos elementos do mesmo tipo
- **Heterogêneas:** Elementos de tipos diferentes

2. Arrays (Vetores) - Estruturas Homogêneas

Definição Matemática

Um array é uma função que mapeia índices para valores:

$$A : \{0, 1, 2, \ldots, n-1\} \to T$$

Onde T é o tipo dos elementos.

Propriedades:

- **Acesso:** $O(1)$ por índice
- **Inserção/Remoção:** $O(n)$ (worst case)
- **Busca:** $O(n)$ linear, $O(\log n)$ se ordenado

Declaração de Arrays em C

Sintaxe Básica

```
tipo nome[tamanho];

// Exemplos
int numeros[10];           // Array de 10 inteiros
float notas[5];            // Array de 5 floats
char texto[100];           // String (array de chars)
```

Inicialização

```
int arr1[5] = {1, 2, 3, 4, 5};      // Inicialização completa
int arr2[5] = {1, 2};               // Parcial: {1, 2, 0, 0, 0}
int arr3[] = {1, 2, 3};             // Tamanho automático: 3
```

Operações Fundamentais em Arrays

1. Acesso e Modificação

```
int arr[5] = {10, 20, 30, 40, 50};

// Acesso: O(1)
int valor = arr[2];     // valor = 30

// Modificação: O(1)
arr[2] = 35;            // arr[2] agora é 35
```

Cálculo de Endereço

$$\text{endereco}(arr[i]) = \text{base} + i \times \text{sizeof}(\text{tipo})$$

2. Algoritmos Básicos em Arrays

Busca Linear

```c
int busca_linear(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) {
            return i;  // Retorna índice
        }
    }
    return -1;  // Não encontrado
}
```

Complexidade: $T(n) = O(n)$
Melhor caso: $O(1)$ (primeiro elemento)
Pior caso: $O(n)$ (último ou não existe)

Busca Binária (Array Ordenado)

```c
int busca_binaria(int arr[], int l, int r, int x) {
    while (l <= r) {
        int m = l + (r - l) / 2;

        if (arr[m] == x) return m;

        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}
```

Complexidade: $T(n) = O(\log n)$
Recorrência: $T(n) = T(n/2) + O(1)$

3. Inserção e Remoção em Arrays

Inserção no Final

```c
int inserir_final(int arr[], int *tamanho, int elemento, int capacidade) {
    if (*tamanho >= capacidade) return 0;  // Array cheio

    arr[*tamanho] = elemento;
    (*tamanho)++;
    return 1;  // Sucesso
}
```

Complexidade: $O(1)$

Inserção em Posição Específica

```c
int inserir_posicao(int arr[], int *tamanho, int pos, int elemento, int cap) {
    if (*tamanho >= cap || pos > *tamanho) return 0;

    // Desloca elementos para direita
    for (int i = *tamanho; i > pos; i--) {
        arr[i] = arr[i-1];
    }

    arr[pos] = elemento;
    (*tamanho)++;
    return 1;
}
```

Complexidade: $O(n)$ - devido ao deslocamento

Remoção de Elementos

Remoção por Índice

```c
int remover_indice(int arr[], int *tamanho, int indice) {
    if (indice >= *tamanho || indice < 0) return 0;

    // Desloca elementos para esquerda
    for (int i = indice; i < *tamanho - 1; i++) {
        arr[i] = arr[i + 1];
    }

    (*tamanho)--;
    return 1;
}
```

Complexidade: $O(n)$

4. Matrizes (Arrays Bidimensionais)

Definição Matemática

Uma matriz é uma função:

$$M : \{0, 1, \ldots, m-1\} \times \{0, 1, \ldots, n-1\} \to T$$

Representação na Memória

Row-major (C): $M[i][j] \to base + (i \times cols + j) \times sizeof(T)$

Column-major (Fortran): $M[i][j] \to base + (j \times rows + i) \times sizeof(T)$

Declaração de Matrizes em C

```c
// Matriz estática
int matriz[3][4];  // 3 linhas, 4 colunas

// Inicialização
int mat[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Alocação dinâmica
int **matriz_dinamica = malloc(linhas * sizeof(int*));
for (int i = 0; i < linhas; i++) {
    matriz_dinamica[i] = malloc(colunas * sizeof(int));
}
```

Operações com Matrizes

Multiplicação de Matrizes

$$C[i][j] = \sum_{k=0}^{p-1} A[i][k] \times B[k][j]$$

```
void multiplicar_matrizes(int A[][3], int B[][3], int C[][3], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Complexidade: $O(n^3)$

Transposição de Matriz

Conceito Matemático

$$A^T[j][i] = A[i][j]$$

```
void transpor_matriz(int A[][MAX], int T[][MAX], int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            T[j][i] = A[i][j];
        }
    }
}
```

Complexidade: $O(m \times n)$

5. Ponteiros: Conceitos Fundamentais

Definição

Um ponteiro é uma variável que armazena o endereço de outra variável:

$$ptr : \text{Endereço} \rightarrow \text{Valor}$$

Declaração e Uso

```c
int x = 42;
int *ptr = &x;    // ptr aponta para x

printf("%d\n", x);     // 42 (valor de x)
printf("%p\n", &x);    // Endereço de x
printf("%p\n", ptr);   // Endereço de x (mesmo que &x)
printf("%d\n", *ptr);  // 42 (valor apontado por ptr)
```

Aritmética de Ponteiros

Operações Válidas

```c
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;  // p aponta para arr[0]

printf("%d\n", *p);       // 10
printf("%d\n", *(p+1));   // 20
printf("%d\n", *(p+2));   // 30

p++;                      // p agora aponta para arr[1]
printf("%d\n", *p);       // 20
```

Cálculo de Endereços

$$p + i = p + i \times sizeof(\text{tipo})$$

Ponteiros e Arrays

Equivalência

```c
int arr[5] = {1, 2, 3, 4, 5};

// Estas expressões são equivalentes:
arr[i]    ≡    *(arr + i)
&arr[i]   ≡    arr + i
```

Passagem para Funções

```c
void processar_array(int *arr, int tamanho) {
    // ou: void processar_array(int arr[], int tamanho)
    for (int i = 0; i < tamanho; i++) {
        arr[i] *= 2;  // Modifica array original
    }
}
```

6. Alocação Dinâmica de Memória

Funções Básicas

```c
#include <stdlib.h>

// Alocação
void *malloc(size_t tamanho);
void *calloc(size_t num, size_t tamanho);
void *realloc(void *ptr, size_t novo_tamanho);

// Liberação
void free(void *ptr);
```

Exemplo: Array Dinâmico

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

typedef struct {
    int *dados;
    int tamanho;
    int capacidade;
} ArrayDinamico;

ArrayDinamico* criar_array(int capacidade_inicial) {
    ArrayDinamico *arr = malloc(sizeof(ArrayDinamico));
    arr->dados = malloc(capacidade_inicial * sizeof(int));
    arr->tamanho = 0;
    arr->capacidade = capacidade_inicial;
    return arr;
}

void liberar_array(ArrayDinamico *arr) {
    free(arr->dados);
    free(arr);
}
```

Redimensionamento Automático

```c
int adicionar_elemento(ArrayDinamico *arr, int elemento) {
    if (arr->tamanho >= arr->capacidade) {
        // Dobra a capacidade
        int nova_capacidade = arr->capacidade * 2;
        int *novos_dados = realloc(arr->dados,
                                   nova_capacidade * sizeof(int));

        if (novos_dados == NULL) return 0;  // Erro

        arr->dados = novos_dados;
        arr->capacidade = nova_capacidade;
    }

    arr->dados[arr->tamanho++] = elemento;
    return 1;
}
```

Complexidade Amortizada: $O(1)$

7. Estruturas Heterogêneas (Structs)

Definição

Uma struct agrupa dados de tipos diferentes:

```
struct Pessoa {
    char nome[50];
    int idade;
    float altura;
    char sexo;
};

// Usando typedef
typedef struct {
    int x, y;
} Ponto;
```

Operações com Structs

Declaração e Inicialização

```c
// Declaração
struct Pessoa p1;
Ponto origem = {0, 0};

// Inicialização designada (C99)
struct Pessoa p2 = {
    .nome = "João",
    .idade = 25,
    .altura = 1.75,
    .sexo = 'M'
};
```

Acesso aos Membros

```c
p1.idade = 30;
strcpy(p1.nome, "Maria");

// Com ponteiros
struct Pessoa *ptr = &p1;
ptr->idade = 35;  // Equivale a (*ptr).idade = 35;
```

Struct com Arrays e Ponteiros

```c
typedef struct {
    int *notas;
    int num_notas;
    char nome[50];
    float media;
} Estudante;

void calcular_media(Estudante *e) {
    int soma = 0;
    for (int i = 0; i < e->num_notas; i++) {
        soma += e->notas[i];
    }
    e->media = (float)soma / e->num_notas;
}
```

8. Unions - Compartilhamento de Memória

Conceito

Uma union permite que diferentes tipos compartilhem a mesma área de memória:

```c
union Valor {
    int inteiro;
    float real;
    char caractere;
};

union Valor v;
v.inteiro = 42;
printf("%d\n", v.inteiro);    // 42

v.real = 3.14;
printf("%f\n", v.real);       // 3.14
// v.inteiro agora tem valor indefinido
```

Exemplo Prático: Sistema de Tipos

```c
typedef enum {
    TIPO_INT,
    TIPO_FLOAT,
    TIPO_STRING
} TipoDado;

typedef struct {
    TipoDado tipo;
    union {
        int valor_int;
        float valor_float;
        char valor_string[100];
    } dados;
} Variavel;

void imprimir_variavel(Variavel *v) {
    switch (v->tipo) {
        case TIPO_INT:
            printf("%d\n", v->dados.valor_int);
            break;
        case TIPO_FLOAT:
            printf("%.2f\n", v->dados.valor_float);
            break;
        case TIPO_STRING:
            printf("%s\n", v->dados.valor_string);
            break;
    }
}
```

9. Comparação C vs Python

Arrays em Python (Lists)

```
# Lista dinâmica
numeros = [1, 2, 3, 4, 5]
numeros.append(6)        # O(1) amortizado
numeros.insert(2, 10)    # O(n)
del numeros[1]           # O(n)

# List comprehension
quadrados = [x**2 for x in range(10)]

# Slicing
sub_lista = numeros[1:4]  # [2, 10, 4]
```

Numpy Arrays (Homogêneos)

```
import numpy as np

# Array homogêneo
arr = np.array([1, 2, 3, 4, 5])
matriz = np.array([[1, 2], [3, 4]])

# Operações vetorizadas
resultado = arr * 2          # [2, 4, 6, 8, 10]
produto = matriz @ matriz    # Multiplicação matricial
```

10. Análise de Complexidade

Comparação de Operações

Operação	Array C	Python List	Numpy Array
Acesso	$O(1)$	$O(1)$	$O(1)$
Busca	$O(n)$	$O(n)$	$O(n)$
Inserção final	$O(1)$	$O(1)$ amort.	$O(n)$
Inserção meio	$O(n)$	$O(n)$	$O(n)$
Remoção	$O(n)$	$O(n)$	$O(n)$

Consumo de Memória

C - Array Estático

```c
int arr[1000];  // 4000 bytes (exato)
```

C - Struct

```c
struct Exemplo {
    char c;     // 1 byte
    int i;      // 4 bytes
    double d;   // 8 bytes
};  // Total: pode ser 16 bytes (com padding)
```

Padding e Alinhamento

$$\text{sizeof(struct)} \geq \sum \text{sizeof(membros)}$$

11. Algoritmos de Ordenação

Bubble Sort

```c
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int houve_troca = 0;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                trocar(&arr[j], &arr[j+1]);
                houve_troca = 1;
            }
        }
        if (!houve_troca) break;  // Otimização
    }
}
```

Complexidade:

- Melhor caso: $O(n)$ (já ordenado)
- Caso médio/pior: $O(n^2)$

Selection Sort

```c
void selection_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int min_idx = i;

        // Encontra o menor elemento
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Troca se necessário
        if (min_idx != i) {
            trocar(&arr[i], &arr[min_idx]);
        }
    }
}
```

Complexidade: $O(n^2)$ sempre

12. Strings em C

Representação

```c
char str1[20] = "Hello";        // Array de caracteres
char *str2 = "World";           // Ponteiro para literal
char str3[] = {'H','i','\0'};   // Inicialização char por char
```

Funções da string.h

```c
#include <string.h>

strlen(str);           // Comprimento
strcpy(dest, src);     // Cópia
strcat(dest, src);     // Concatenação
strcmp(str1, str2);    // Comparação
```

Implementação de strlen

```c
size_t meu_strlen(const char *str) {
    size_t len = 0;
    while (str[len] != '\0') {
        len++;
    }
    return len;
}

// Versão com ponteiro
size_t strlen_ptr(const char *str) {
    const char *inicio = str;
    while (*str) str++;
    return str - inicio;
}
```

Complexidade: $O(n)$ onde n é o comprimento da string

13. Estruturas de Dados Avançadas

Lista Ligada com Struct

```c
typedef struct No {
    int dado;
    struct No *proximo;
} No;

typedef struct {
    No *cabeca;
    int tamanho;
} Lista;

void inserir_inicio(Lista *lista, int valor) {
    No *novo = malloc(sizeof(No));
    novo->dado = valor;
    novo->proximo = lista->cabeca;
    lista->cabeca = novo;
    lista->tamanho++;
}
```

Pilha (Stack) com Array

```c
#define MAX_SIZE 100

typedef struct {
    int dados[MAX_SIZE];
    int topo;
} Pilha;

void push(Pilha *p, int valor) {
    if (p->topo < MAX_SIZE - 1) {
        p->dados[++p->topo] = valor;
    }
}

int pop(Pilha *p) {
    if (p->topo >= 0) {
        return p->dados[p->topo--];
    }
    return -1;  // Pilha vazia
}
```

Complexidade: $O(1)$ para push e pop

14. Aplicações Práticas

Sistema de Cadastro

```c
typedef struct {
    int id;
    char nome[50];
    char email[100];
    float salario;
} Funcionario;

typedef struct {
    Funcionario *funcionarios;
    int quantidade;
    int capacidade;
} BaseDados;

int buscar_por_id(BaseDados *db, int id) {
    for (int i = 0; i < db->quantidade; i++) {
        if (db->funcionarios[i].id == id) {
            return i;
        }
    }
    return -1;
}
```

Matriz Esparsa

```c
typedef struct {
    int linha;
    int coluna;
    double valor;
} Elemento;

typedef struct {
    Elemento *elementos;
    int num_elementos;
    int linhas;
    int colunas;
} MatrizEsparsa;

void adicionar_elemento(MatrizEsparsa *m, int i, int j, double valor) {
    if (valor != 0.0) {
        m->elementos[m->num_elementos++] = (Elemento){i, j, valor};
    }
}
```

15. Otimizações e Considerações

Cache Locality

```
// Bom para cache (row-major)
for (int i = 0; i < ROWS; i++) {
    for (int j = 0; j < COLS; j++) {
        matriz[i][j] = i + j;
    }
}

// Ruim para cache (column-major em C)
for (int j = 0; j < COLS; j++) {
    for (int i = 0; i < ROWS; i++) {
        matriz[i][j] = i + j;  // Acesso não sequencial
    }
}
```

Memory Alignment

```
// Sem padding otimizado
struct Ineficiente {
    char a;     // 1 byte
    double b;   // 8 bytes (7 bytes de padding)
    char c;     // 1 byte (7 bytes de padding no final)
};  // Total: 24 bytes

// Com padding otimizado
struct Eficiente {
    double b;   // 8 bytes
    char a;     // 1 byte
    char c;     // 1 byte (6 bytes de padding)
};  // Total: 16 bytes
```

16. Depuração e Testes

Validação de Ponteiros

```c
void funcao_segura(int *ptr) {
    if (ptr == NULL) {
        printf("Erro: ponteiro nulo!\n");
        return;
    }

    // Usar o ponteiro seguramente
    *ptr = 42;
}
```

Detecção de Memory Leaks

```c
void testar_memoria() {
    int *arr = malloc(100 * sizeof(int));

    // ... usar array ...

    free(arr);  // IMPORTANTE: sempre liberar
    arr = NULL; // Boa prática
}
```

17. Benchmarking

Medição de Performance

```c
#include <time.h>

void benchmark_algoritmo() {
    clock_t inicio = clock();

    // Algoritmo a ser testado
    for (int i = 0; i < 1000000; i++) {
        // Operação repetitiva
    }

    clock_t fim = clock();
    double tempo = ((double)(fim - inicio)) / CLOCKS_PER_SEC;
    printf("Tempo: %.6f segundos\n", tempo);
}
```

18. Padrões de Design

Factory Pattern para Structs

```c
Pessoa* criar_pessoa(const char *nome, int idade) {
    Pessoa *p = malloc(sizeof(Pessoa));
    if (p != NULL) {
        strncpy(p->nome, nome, sizeof(p->nome) - 1);
        p->nome[sizeof(p->nome) - 1] = '\0';
        p->idade = idade;
    }
    return p;
}

void destruir_pessoa(Pessoa *p) {
    free(p);
}
```

19. Comparação Final: C vs Python

Vantagens do C:

- **Performance:** 10-100x mais rápido
- **Controle de memória:** Gestão precisa
- **Previsibilidade:** Comportamento determinístico
- **Eficiência espacial:** Menor overhead

Vantagens do Python:

- **Produtividade:** Desenvolvimento mais rápido
- **Flexibilidade:** Tipos dinâmicos
- **Bibliotecas:** NumPy, SciPy, Pandas
- **Expressividade:** Código mais conciso

20. Conclusões e Próximos Passos

O que Aprendemos:

- **Estruturas homogêneas** (arrays) e heterogêneas (structs)
- **Ponteiros** e gerenciamento de memória
- **Algoritmos** fundamentais em estruturas
- **Análise de complexidade** e otimizações
- **Comparações** entre linguagens

Próxima Aula:

- **Análise de Algoritmos** e complexidade computacional
- **Notações assintóticas** avançadas
- **Técnicas de análise** matemática
- **Casos práticos** de otimização

Exercícios Propostos

1. Implemente uma calculadora de matrizes completa
2. Crie um sistema de gerenciamento de estudantes usando structs
3. Desenvolva um array dinâmico genérico (void*)
4. Compare performance: array estático vs dinâmico vs Python list
5. Implemente ordenação eficiente para structs

Bibliografia

- **Cormen, T. H.** et al. *Introduction to Algorithms*, 4ª ed.
- **Kernighan, B. W.; Ritchie, D. M.** *The C Programming Language*, 2ª ed.
- **Sedgewick, R.** *Algorithms in C*, 3ª ed.
- **Tanenbaum, A. S.** *Structured Computer Organization*, 6ª ed.

Contato e Dúvidas

Prof. Vagner Cordeiro
📧 **Email:** [email do professor]
🕐 **Atendimento:** [horários de atendimento]
📚 **Material:** github.com/cordeirotelecom/algoritimos_e_complexidade

Próxima aula: Análise de Algoritmos e Prática de Análise

Obrigado!

Perguntas?

Algoritmos e Complexidade - Aula 02

Estruturas de Dados - Homogêneas, Heterogêneas e Ponteiros