

# Algoritmos e Estruturas de Dados

Tecnólogo em Análise e Desenvolvimento de Sistemas

Prof. Dr. Waldeyr Mendes Cordeiro da Silva

2019

## **Material didático para Algoritmos e Estruturas de Dados**

Versão 0.1

### **Autor: Waldeyr Mendes Cordeiro da Silva**

- Formação:
  - Técnico em Processamento de Dados
  - Bacharelado em Sistemas de Informação
  - Licenciatura em Ciências Biológicas
  - Complementação Pedagógica em Matemática
  - Especialização em Engenharia de Software
  - Especialização em Segurança da Informação
  - Mestrado em Informática
  - Doutorado em Ciências Biológicas (Bioinformática)
-  Lattes: <http://lattes.cnpq.br/2391349697609978>
-  ORCID: <https://orcid.org/0000-0002-8660-6331>
-  Scopus: <https://www.scopus.com/authid/detail.uri?authorId=57190660358>
-  Google Acadêmico: <https://scholar.google.com.br/citations?user=5gh-liIAAAAJ>

# Sumário

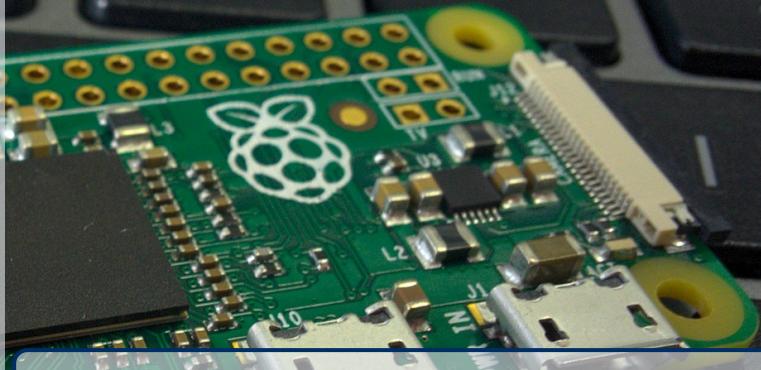
|            |  |    |
|------------|--|----|
| <b>1</b>   | <b>Prefácio</b>                                      | 5  |
| <b>2</b>   | <b>Introdução</b>                                    | 6  |
| <b>2.1</b> | <b>Análise de Algoritmos</b>                         | 7  |
| 2.1.1      | Elementos de Notação Assintótica                     | 7  |
| <b>3</b>   | <b>Programação</b>                                   | 8  |
| <b>3.1</b> | <b>Primeiros Passos em Programação</b>               | 9  |
| <b>3.2</b> | <b>Trabalhando com Variáveis</b>                     | 10 |
| <b>3.3</b> | <b>Estruturas de Decisão</b>                         | 13 |
| 3.3.1      | If/Else  | 13 |
| <b>4</b>   | <b>Estruturas de Dados</b>                           | 15 |
| <b>4.1</b> | <b>Algoritmos de Ordenação</b>                       | 15 |
| <b>4.2</b> | <b>Estruturas de Dados Homogêneas e Heterogêneas</b> | 24 |
| <b>4.3</b> | <b>Listas Encadeadas</b>                             | 27 |
| <b>4.4</b> | <b>Pilhas</b>  | 30 |
| <b>4.5</b> | <b>Filas</b>   | 30 |
| <b>4.6</b> | <b>Árvores</b>                                       | 31 |
| <b>4.7</b> | <b>Tabelas Hashing</b>                               | 32 |

|            |                   |           |
|------------|-------------------|-----------|
| <b>4.8</b> | <b>Grafos</b>     | <b>32</b> |
| 4.8.1      | Busca em Grafos   | 32        |
| <b>5</b>   | <b>Aplicações</b> | <b>33</b> |
| <b>6</b>   | <b>Apêncice</b>   | <b>35</b> |



# 1. Prefácio

Este livro destina-se aos acadêmicos e demais interessados em iniciar os estudos em algoritmos e estruturas de dados.



## 2. Introdução

*Um bonita citação...*

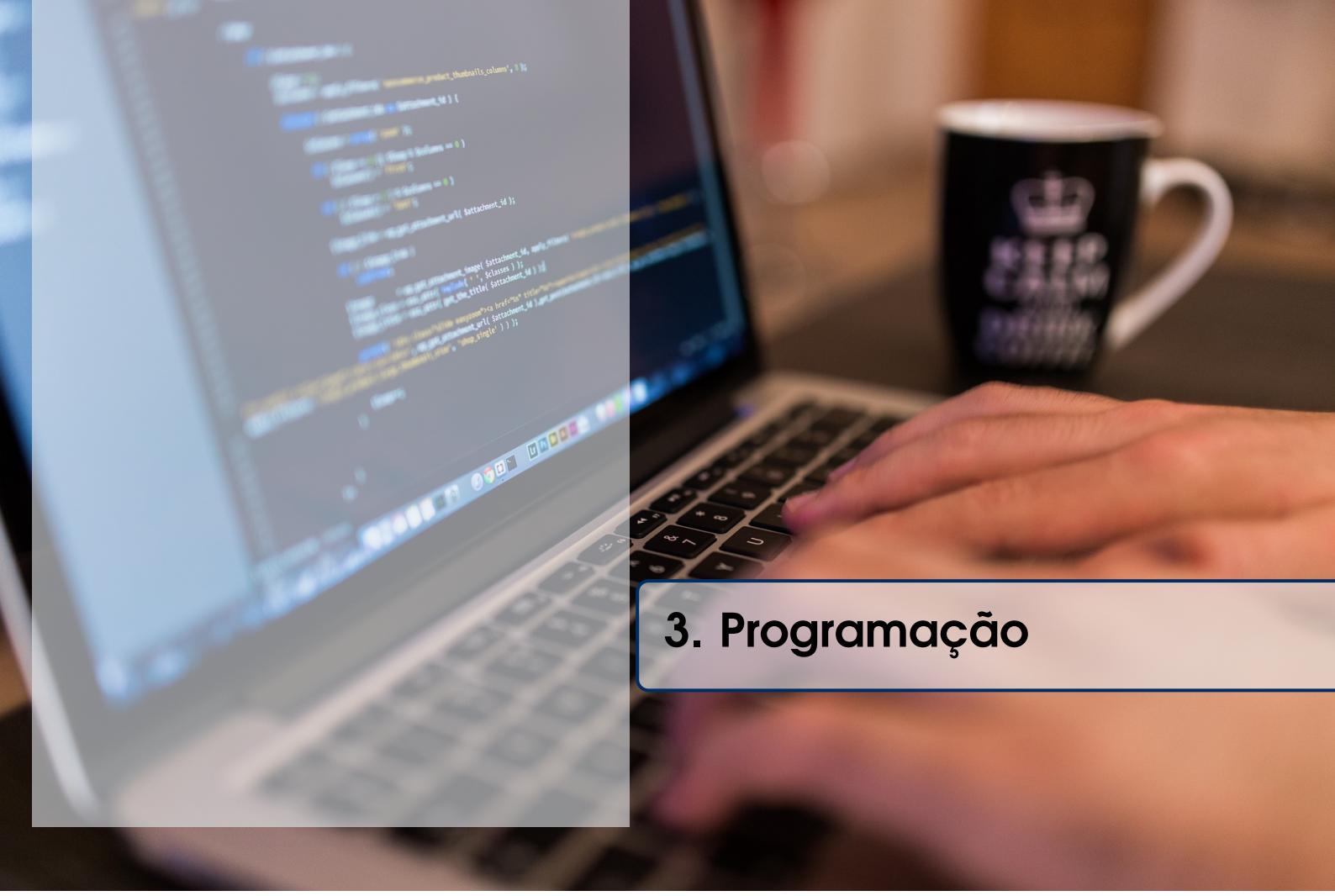
Um algoritmo é um procedimento computacional bem definido que processa um valor ou um conjunto de dados (entrada) e produz algum valor ou conjunto de dados (saída) (CORMEN et al., 2009). Os algoritmos existem há muito tempo, mas estão presentes na sociedade moderna de uma forma nunca experimentada na história da humanidade. Quase tudo que se produz, sejam produtos ou serviços, tem alguma influência de algoritmos. O comércio eletrônico utiliza tanto algoritmos clássicos como ordenações, quanto algoritmos modernos de recomendação de produtos baseado no histórico de visitas. A segurança das senhas em qualquer sistema bancário ou *Web* é garantida por algoritmos de criptografia. Imagens de satélite, dados genômicos, reconhecimento de faces, previsão do tempo, quase tudo que se possa imaginar atualmente é influenciado direta ou indiretamente por algum algoritmo.

O estudo de algoritmos é uma demanda crescente frente aos novos desafios trazidos pelo grande volume de dados que as tecnologias modernas proporcionaram. A análise de algoritmos é uma área com questões importantes em aberto, como é o caso dos *Millennium Prize Problems*, com sete problemas matemático-computacionais, cujo prêmio é de 1 milhão de dólares pela solução de cada problema.

## 2.1 Análise de Algoritmos

O propósito da análise de um algoritmo é prever seu comportamento quanto ao tempo de execução ou ao espaço em memória que irá ocupar mesmo antes de ser executado em um computador específico. Porém, muitos fatores, como o tamanho e a variedade dos dados de entrada, influenciam o algoritmo. Portanto, a análise de algoritmos provê uma aproximação, o que em muitos casos é bastante significante.

### 2.1.1 Elementos de Notação Assintótica



### 3. Programação

### 3.1 Primeiros Passos em Programação

Dennis Ritchie criou a linguagem “C” em 1972 quando trabalhava nos laboratórios da Bell Telephone. A linguagem C foi criada para desenvolver uma nova versão do sistema operacional Unix que utilizava Assembly até então. Inicialmente, a cada nova implementação de Unix para um tipo de máquina, um novo compilador C devia ser desenvolvido para essa máquina. Nos anos 80, C tornou-se popular fora do ambiente Unix, criando a demanda por compiladores comerciais. Desde então, C é considerada uma linguagem de propósito geral, com instruções de alto nível e capaz de gerar programas rápidos em tempo de execução. Além disso, C influenciou de forma direta a sintaxe e estruturas de linguagens modernas como C++, Java, C#, Objective C, e muitas outras.

A linguagem C começou a ser padronizada pela *American National Standards Institute* (ANSI) para garantir a compatibilidade e a portabilidade da linguagem. Este padrão foi atualizado ao longo do tempo e posteriormente reconhecido pela ISO, dando origem ao padrão ANSI/ISO.

Um código escrito em linguagem C funciona com comandos, que são normalmente palavras em inglês (sintaxe) e são organizados em determinadas estruturas. Uma vez escritos em um arquivo, os comandos são então lidos pelos compiladores e convertidos para a linguagem de máquina que corresponde a ação pretendida. O Programa 3.1 mostra um exemplo de programa em C.

Programa 3.1: Meu primeiro programa em C.

```
1 #include <stdio.h>
2 void main() {
3     printf("Meu primeiro programa em C!\n");
4 }
```

Após escrito, o programa precisa ser compilado. Se o programa tiver sido escrito corretamente (sintaxe e estrutura), o compilador gera um executável. É importante perceber, que compiladores não verificam erros lógicos, ou seja de concepção do código. É justamente aí que reside o valor do trabalho humano como programador: combinar sintaxe e estruturas de forma que solucionem um dado problema do mundo real. Caso o compilador encontre algum erro no código dois tipos de mensagens podem ser emitidas:

1. Erro (*error*): Quando o código tem um problema que impede o compilador de gerar um executável
2. Aviso (*warnning*): Quando o problema no código não impede de gerar um executável, mas requer algum tipo de atenção.

Para compilar o Programa 3.1 em Linux com o compilador C (Veja aqui como instalar) basta o seguinte comando no terminal:

```
gcc prog001.c -o prog001.sh
```

Após esta ação, será gerado o arquivo executável `prog001.sh`, o qual poderá ser executado com o comando:

```
./prog001.sh
```

No Programa 3.1 é impressa uma mensagem na tela. O caractere especial representado por “\n” identifica uma quebra de linha. Essa mensagem é representada como uma cadeira de caracteres (*string*) limitada pelas aspas. Se for necessário representar as aspas em uma *string*, será necessário escapá-las usando a barra “\”. O Programa 3.2 mostra como fazer isso.

Programa 3.2: Escapando aspas.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void main() {
```

---

```
5     printf("Que lindo dia para aprender \"C\"\n");
6 }
```

---

## 3.2 Trabalhando com Variáveis

Uma variável é um objeto capaz de guardar e representar um valor ou expressão em memória. As variáveis são associadas a nomes ou identificadores e sua existência é limitada por um espaço e um tempo na memória do computador, o qual frequentemente é igual ao tempo de execução do programa em que elas foram criadas.

Nas linguagens de programação é comum existirem tipos de dados pré-definidos, chamados tipo primitivos. Em C, os tipos primitivos são:

- char: caractere
- int: inteiro
- float: real de precisão simples
- double: real de alta precisão
- void: vazio ou sem valor

Há ainda os modificadores que alteram o tamanho, e consequentemente sua capacidade, dos tipos em memória. São eles:

- signed
- unsigned
- long
- short

Combinando os tipos e os modificadores temos:

Table 3.1:

| Palavra chave      | Tipo                                     | Tamanho (Bytes) | Intervalo                      |
|--------------------|--|-----------------|--------------------------------|
| char               | Caractere                                | 1               | -128 a 127                     |
| signed char        | Caractere com sinal                      | 1               | -128 a 127                     |
| unsigned char      | Caractere sem sinal                      | 1               | 0 a 255                        |
| int                | Inteiro                                  | 4               | -32.768 a 32.767               |
| signed int         | Inteiro com sinal                        | 4               | -32.768 a 32.767               |
| unsigned int       | Inteiro sem sinal                        | 4               | 0 a 65.535                     |
| short int          | Inteiro curto                            | 2               | -32.768 a 32.767               |
| signed short int   | Inteiro curto com sinal                  | 2               | -32.768 a 32.767               |
| unsigned short int | Inteiro curto sem sinal                  | 2               | 0 a 65.535                     |
| long int           | Inteiro longo                            | 8               | -2.147.483.648 a 2.147.483.647 |
| signed long int    | Inteiro longo com sinal                  | 8               | -2.147.483.648 a 2.147.483.647 |
| unsigned long int  | Inteiro longo sem sinal                  | 8               | 0 a 4.294.967.295              |
| float              | Ponto flutuante com precisão simples     | 4               | 3.4 E-38 a 3.4E+38             |
| double             | Ponto flutuante com precisão simples     | 8               | 1.7 E-308 a 1.7E+308           |
| long double        | Ponto flutuante com precisão dupla longo | 16              | 3.4E-4932 a 1.1E+4932          |

O Programa 3.3 mostra um exemplo de variável representando o valor inteiro 10 e em seguida a impressão de uma mensagem com seu valor. Na impressão, é importante notar que a mensagem é formatada para que em um determinado ponto (\d) apareçam os dígitos correspondentes ao valor de a que é a variável abrigando o valor 10.

Programa 3.3: Trabalhando com variáveis.

---

```
1 #include <stdio.h>
2
3 void main() {
4     int a = 10;
5     printf("O valor de a = \t%d\n", a);
6 }
```

---

As formas de declarar variáveis podem ser diversas. O Programa 3.4 mostra a declaração de variáveis *n1* e *n2* atribuindo imediatamente os valores, enquanto a variável *soma* é criada sem que seu valor seja atribuído imediatamente. No Programa 3.5, as variáveis *n1* e *n2* são declaradas na mesma linha, separadas por vírgula, uma vez que são do mesmo tipo *int*.

Programa 3.4: Trabalhando com variáveis.

---

```

1 #include <stdio.h>
2 void main() {
3     int n1 = 37;
4     int n2 = -37;
5     int soma;
6     soma = n1 + n2;
7     printf("%d + %d = %d \n", n1, n2, soma);
8 }
```

---

Programa 3.5: Trabalhando com variáveis.

---

```

1 #include <stdio.h>
2
3 void main() {
4     int n1 = 1, n2 = 2;
5     int soma = n1 + n2;
6     printf("%d + %d = %d \n", n1, n2, soma);
7 }
```

---

Quanto à atribuição de valores à uma variável, ela pode ser direta ou indireta, ou seja, um valor pode ser diretamente atribuído à uma variável ou o valor de uma variável pode ser atribuído a outra, como no exemplo do Programa 3.6, onde *n1* recebe o valor de *n2* que recebe o valor de *n3* que recebe 10. Nesse caso, *n1*, *n2* e *n3* terão valor = 10.

Programa 3.6: Trabalhando com variáveis.

---

```

1 #include <stdio.h>
2
3 void main() {
4     int n1;
5     int n2;
6     int n3;
7     n1 = n2 = n3 = 10;
8     printf("n1 = %d \n", n1);
9     printf("n2 = %d \n", n2);
10    printf("n3 = %d \n", n3);
11 }
```

---

Dependendo do tipo primitivo de dados da variável, diferentes formatações podem ser usadas para a função printf() conforme pode ser visto na Tabela 3.2. O Programa 3.7 ilustra um exemplo de impressão de um float com 2 casas decimais.

Programa 3.7: Trabalhando com variáveis.

---

```

1 #include <stdio.h>
2 void main() {
3     float pi = 3.1415;
4     printf("Pi = %.2f\n", pi);
5 }
6 }
```

---

O Programa 3.8 mostra algumas variáveis e seus respectivos tamanhos em *Bytes* com sua saída formatada.

Programa 3.8: Trabalhando com variáveis.

---

```

1 #include <stdio.h>
2 void main() {
3     signed char v1 = 'A';
4 }
```

---

Table 3.2: Formatação da saída de dados com a função printf().

| Formatação | Significado                 |
|------------|-----------------------------|
| %c         | caracter                    |
| %d         | inteiro base 10             |
| %e         | exponencial ponto flutuante |
| %f         | ponto flutuante             |
| %i         | integer (base 10)           |
| %o         | octal (base 8)              |
| %s         | string                      |
| %u         | <i>unsigned</i> inteiro     |
| %x         | hexadecimal (base 16)       |
| %%         | caracter de porcentagem     |

```

4   int v2 = 2147483647;
5   short v3 = 32767;
6   long int v4 = 2147483647;
7   float v5 = 3.4E38;
8   double v6 = 1.7E+308;
9   long double v7 = 1.79769e+308;
10  printf("Tamanho de %c = %d bytes \n", v1, sizeof(v1) );
11  printf("Tamanho de %d = %d bytes \n", v2, sizeof(v2) );
12  printf("Tamanho de %d = %d bytes \n", v3, sizeof(v3) );
13  printf("Tamanho de %d = %d bytes \n", v4, sizeof(v4) );
14  printf("Tamanho de %.1f = %d bytes \n", v5, sizeof(v5) );
15  printf("Tamanho de %.1f = %d bytes \n", v6, sizeof(v6) );
16  printf("Tamanho de %.1f = %d bytes \n", v7, sizeof(v7) );
17 }
```

### Operadores Aritméticos

Os operadores aritméticos usados em C são mostrados na Tabela 3.3. O Programa 3.9 ilustra o

Table 3.3: Operadores aritméticos.

| Operador                      | Sintaxe |
|-------------------------------|---------|
| Adição unária                 | +a      |
| Adição                        | a + b   |
| Incremento pré-fixado         | ++a     |
| Incremento pós-fixado         | a++     |
| Atribuição por adição         | a += b  |
| Subtração unária              | -a      |
| Subtração                     | a - b   |
| Decremento pré-fixado         | -a      |
| Decremento pós-fixado         | a-      |
| Atribuição por subtração      | a -= b  |
| Multiplicação                 | a * b   |
| Atribuição por multiplicação  | a *= b  |
| Divisão                       | a / b   |
| Atribuição por divisão        | a /= b  |
| Módulo (resto)                | a % b   |
| Atribuição por módulo (resto) | a %= b  |

Table 3.4: Precedência de operadores.

| Prioridade | Operadores  |
|------------|---|
| 1º         | Parênteses internos   |
| 2º         | potência (^) e raiz (quando a linguagem oferece esses operadores) |
| 3º         | * / div e mod   |
| 4º         | + e -   |

uso do operador *mod*, que retorna o resto da divisão inteira.

Programa 3.9: Trabalhando com variáveis.

```

1 #include <stdio.h>
2
3 void main() {
4     // Operators
5     // + soma
6     // - subtracao
7     // * multiplicacao
8     // / divisao
9     // % modulo (divisao inteira)
10    int n1 = 21;
11    int m = 21 % 4;
12    printf("The module of %d is %d \n", n1, m);
13 }
```

### Entrada de Dados

A maneira mais clássica de um programa em C obter dados digitados pelo usuário através do *prompt* é usando a função *scanf()*. O Programa 3.10 mostra um exemplo onde os valores das variáveis *a* e *b* são informados pelo usuário e depois somados.

Programa 3.10: Trabalhando com a função *scanf()* para entrada de dados em C.

```

1 #include <stdio.h>
2 void main(){
3     int a, b, n;
4
5     printf("Digite um numero inteiro: ");
6     scanf("%d", &a);
7
8     printf("Digite um numero inteiro: ");
9     scanf("%d", &b);
10
11    soma = a + b;
12    printf("O valor da soma: %d\n", soma);
13 }
```

## 3.3 Estruturas de Decisão

### 3.3.1 If/Else

O *if* é utilizado para testar uma proposição lógica. A proposição é escrita entre parênteses. Caso o resultado seja verdadeiro, o bloco de instruções subordinadas ao *if* será executado. O bloco é delimitado por “{” e “}”. Se houver apenas uma instrução subordinada, não há necessidade das chaves.

O *else* é utilizado para capturar a condição não atendida no teste do *if*, o “caso contrário”. O Programa 3.11 mostra um exemplo onde o valor da variável *x* é testado para saber se é maior ou igual a zero. Através deste teste é possível identificar se o valor é positivo (*if* verdadeiro  $\rightarrow \text{valor} \geq 0$ ), ou é negativo (*else*  $\rightarrow \text{valor} < 0$ ).

Programa 3.11: Trabalhando com o *if* em C.

---

```
1 #include <stdio.h>
2 void main() {
3     int x;
4     printf("Informe um valor inteiro: ");
5     scanf("%d", &x);
6     if (x>=0) {
7         printf("%d: positivo.\n", x);
8     }
9     else{
10         printf("%d: negativo.\n", x);
11     }
12 }
```

---

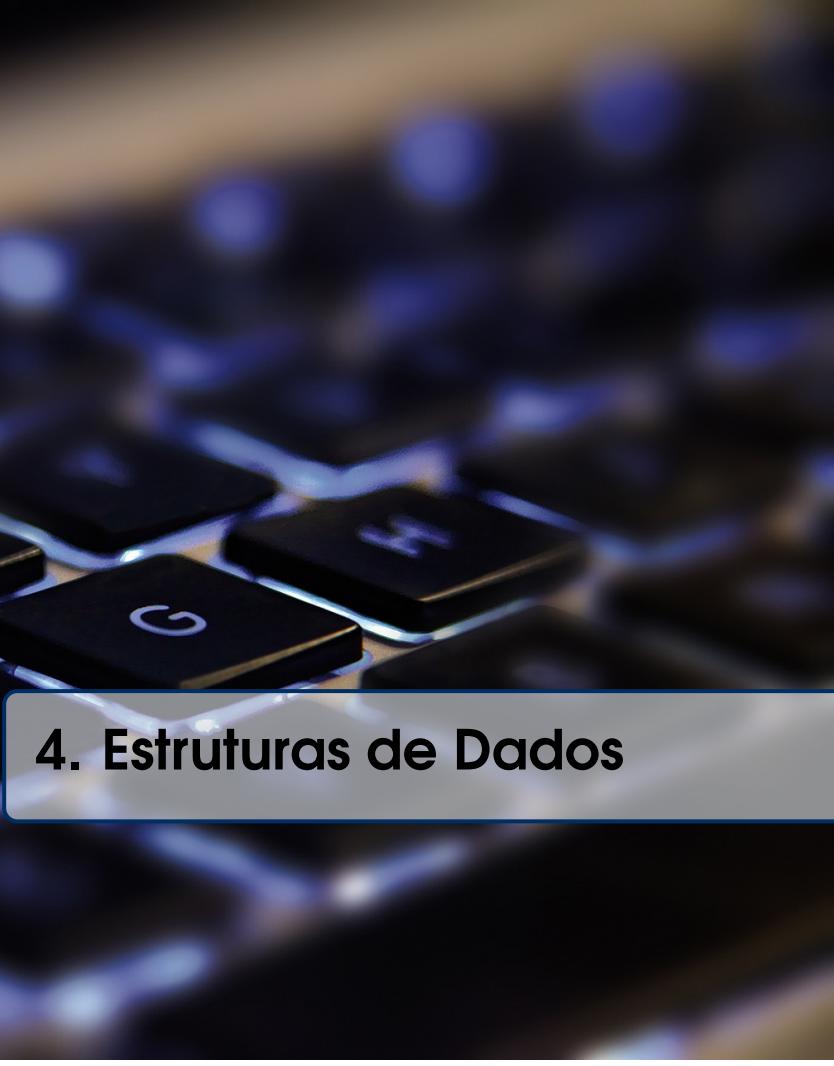
Há ainda a possibilidade verificar múltiplas condições com o *else if*. O uso do *else if* é demonstrado no Programa 3.12 que verifica se um valor inteiro informado pelo usuário é = 10, 20 ou 30.

Programa 3.12: Trabalhando com o *if, else if, else* em C.

---

```
1 #include <stdio.h>
2 void main() {
3     int x;
4     printf("Informe um valor inteiro: ");
5     scanf("%d", &x);
6     if (x==10){
7         printf("%d.\n", x);
8     }
9     else if (x==20){
10        printf("%d.\n", x);
11    }
12    else if(x==30){
13        printf("%d.\n", x);
14    }
15    else{
16        printf("%d: diferente de 10, 20 ou 30.\n");
17    }
18 }
```

---



## 4. Estruturas de Dados

Tipos abstratos de dados são conjuntos de valores sobre os quais é possível aplicar funções de forma homogênea através de um algoritmo. Funções e valores em conjunto, constituem um modelo matemático que pode ser empregado em problemas do mundo real ([ASCENCIO; ARAÚJO, 2010](#)). Os algoritmos são projetados em função de um tipo abstrato de dados. A representação computacional de um tipo abstrato de dados com seus tipos e operações permitidas pode ser entendida como uma **estrutura de dados**. Uma estrutura de dados é um meio para armazenar e processar dados com vistas à sua organização, acesso e modificações ([CORMEN et al., 2009](#)).

### 4.1 Algoritmos de Ordenação

#### Bubble Sort

O algoritmo da bolha (*Bubble Sort*) é um algoritmo de ordenação onde cada elemento de uma posição  $i$  é comparado com o elemento de posição  $i + 1$ , os quais trocam de posição, se for o caso, para atender à ordenação procurada (crescente ou decrescente). O programa [4.1](#) apresenta uma implementação em C do algoritmo *Bubble Sort*, enquanto a Figura [4.1](#) ilustra seu funcionamento

para um vetor de tamanho 4.

Programa 4.1: Implementação em C do algoritmo de ordenação BubbleSort.

---

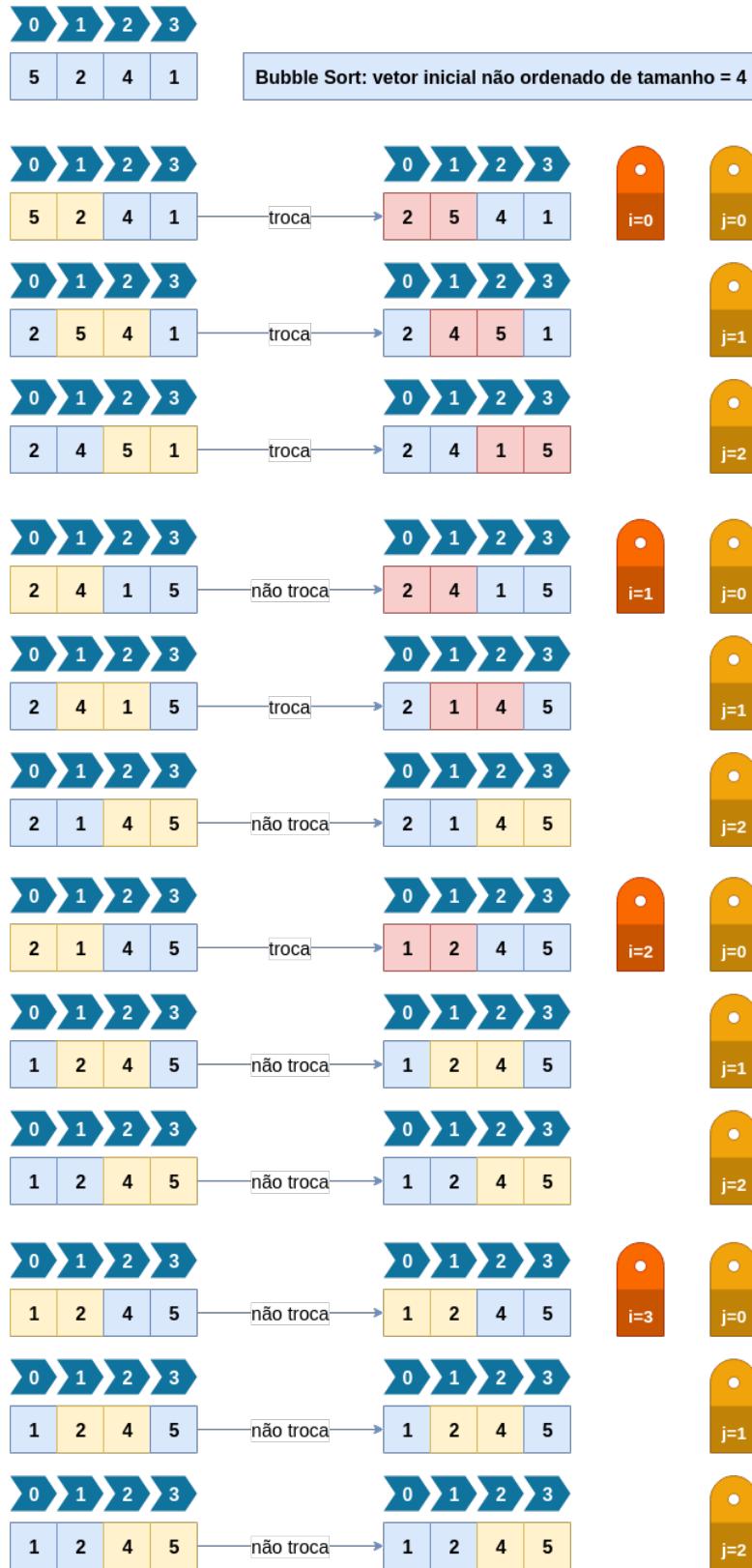
```

1 #include <stdio.h>
2 #include <time.h>
3 #include <stdlib.h>
4 #define TAMANHO 4
5 void main() {
6     int vetor[TAMANHO]; //vetor com tamanho definido
7     int aux = 0; //variavel para ser usada na troca
8     clock_t tempoInicial, tempoFinal; //Variaveis para guardar o tempo de execucao
9     srand(time(NULL)); //Cria uma semente para numeros aleatorios
10    tempoInicial = clock(); //inicia contagem do tempo
11    for (int i = 0; i < TAMANHO; i++) {
12        vetor[i] = rand() % 10; //Atribui um inteiro aleatorio entre 0 e 9
13    }
14    //Mostra valores do vetor nao ordenado
15    for (int i = 0; i < TAMANHO; i++) {
16        printf("%d\t", vetor[i]);
17    }
18    printf("\n");
19    //Ordena vetor pelo metodo da bolha
20    for (int i = 1; i < TAMANHO; i++) {
21        for (int j = 0; j < TAMANHO - 1; j++) {
22            if (vetor[j] > vetor[j + 1]) {
23                aux = vetor[j];
24                vetor[j] = vetor[j + 1];
25                vetor[j + 1] = aux;
26            }
27        }
28    }
29    //Mostra valores do vetor ordenado
30    for (int i = 0; i < TAMANHO; i++) {
31        printf("%d\t", vetor[i]);
32    }
33    printf("\n");
34    tempoFinal = clock(); //finaliza contagem do tempo
35    //calcula e mostra o tempo total de execucao
36    printf("Tempo: %f s\n", (double) (tempoFinal - tempoInicial) / CLOCKS_PER_SEC);
37 }
```

---

## Exercícios

1. Atualizar a implementação do *Bubble Sort* para executar as seguinte tarefa:
  - executar o algoritmo com três vetores de entrada:
    - (a) um vetor ordenado crescente de tamanho 100000 (cem mil)
    - (b) um vetor com números aleatórios de tamanho 100000 (cem mil)
    - (c) um vetor ordenado decrescente de tamanho 100000 (cem mil)
  - criar variáveis para contar a quantidade de comparações e quantidade de trocas;
2. Criar gráficos para as três execuções comparando quantidade de comparações e quantidade de trocas e o tempo de execução;

Fig. 4.1: Algoritmo *Bubble Sort* para um vetor de tamanho 4.

### Selection Sort

No algoritmo de ordenação por seleção (*Selection Sort*) cada número do vetor, a partir do primeiro, é eleito e comparado com o menor número<sup>1</sup> entre aqueles que estão à direita do eleito. O programa 4.1 apresenta uma implementação em C do algoritmo *Selection Sort* para um vetor de tamanho 4.

Programa 4.2: Implementação em C do algoritmo de ordenação Selection.

```

1 #include <stdio.h>
2 #include <time.h>
3 #include <stdlib.h>
4 #define TAMANHO 4
5 void main() {
6     int vetor[TAMANHO]; //vetor com tamanho definido
7     int aux = 0; //varivel para ser usada na troca
8     int eleito, menor, posicaoDoMenor;
9     clock_t tempoInicial, tempoFinal; //Variaveis para guardar o tempo de execucao
10    srand(time(NULL)); //Cria uma semente para numeros aleatorios
11    tempoInicial = clock(); //inicia contagem do tempo
12    for (int i = 0; i < TAMANHO; i++) {
13        vetor[i] = rand() % 10; //Atribui um inteiro aleatorio entre 0 e 9
14    }
15    //Mostra valores do vetor nao ordenado
16    for (int i = 0; i < TAMANHO; i++) {
17        printf("%d\t", vetor[i]);
18    }
19    printf("\n");
20    //Ordena vetor pelo metodo da da selecao
21    for (int i = 0; i < TAMANHO-2; i++) { // 0 -> penultima posicao
22        eleito = vetor[i];
23        menor = vetor[i+1];
24        posicaoDoMenor = i+1;
25        for (int j = i+1; j < TAMANHO; j++) {
26            if (vetor[j] < menor) {
27                menor = vetor[j];
28                posicaoDoMenor = j;
29            }
30        }
31        if( menor < eleito ){
32            vetor[i] = vetor[posicaoDoMenor];
33            vetor[posicaoDoMenor] = eleito;
34        }
35    }
36    //Mostra valores do vetor ordenado
37    for (int i = 0; i < TAMANHO; i++) {
38        printf("%d\t", vetor[i]);
39    }
40    printf("\n");
41    tempoFinal = clock(); //finaliza contagem do tempo
42    //calcula e mostra o tempo total de execucao
43    printf("Tempo: %f s\n", (double) (tempoFinal - tempoInicial) / CLOCKS_PER_SEC);
44 }
```

### Exercícios

1. Atualizar a implementação do *Selection Sort* para executar as seguinte tarefa:
  - executar o algoritmo com três vetores de entrada:
    - (a) um vetor ordenado crescente de tamanho 100000 (cem mil)
    - (b) um vetor com números aleatórios de tamanho 100000 (cem mil)
    - (c) um vetor ordenado decrescente de tamanho 100000 (cem mil)
  - criar variáveis para contar a quantidade de comparações e quantidade de trocas;
2. Criar gráficos para as três execuções comparando quantidade de comparações e quantidade de trocas e o tempo de execução;

<sup>1</sup>Ou maior dependendo da ordenação desejada.

### Insertion Sort

O algoritmo de ordenação por inserção (*Insertion Sort*), funciona como uma mão de baralho onde você ordena suas cartas começando a comparação entre as duas primeiras. No *Insertion Sort*, é eleito o segundo elemento do vetor para iniciar a comparação. É percorrido o vetor de forma a garantir que todos os números à esquerda do eleito sejam menores que ele. O programa 4.1 apresenta uma implementação em C do algoritmo *Insertion Sort* para um vetor de tamanho 4.

Programa 4.3: Implementação em C do algoritmo de ordenação Insertion Sort.

---

```

1 #include <stdio.h>
2 #include <time.h>
3 #include <stdlib.h>
4 #define TAMANHO 4
5 void main() {
6     int vetor[TAMANHO]; //vetor com tamanho definido
7     int eleito = 0;
8     int j = 0;
9     clock_t tempoInicial, tempoFinal; //Variaveis para guardar o tempo de execucao
10    srand(time(NULL)); //Cria uma semente para numeros aleatorios
11    tempoInicial = clock(); //inicia contagem do tempo
12    for (int i = 0; i < TAMANHO; i++) {
13        vetor[i] = rand() % 10; //Atribui um inteiro aleatorio entre 0 e 9
14    }
15    //Mostra valores do vetor nao ordenado
16    for (int i = 0; i < TAMANHO; i++) {
17        printf("%d\t", vetor[i]);
18    }
19    printf("\n");
20    //Ordena vetor pelo metodo da selecao
21    for (int i = 1; i < TAMANHO; i++) {
22        eleito = vetor[i];
23        j = i - 1;
24        while (j >= 0 && vetor[j] > eleito) {
25            vetor[j + 1] = vetor[j];
26            j--;
27        }
28        vetor[j + 1] = eleito;
29    }
30    //Mostra valores do vetor ordenado
31    for (int i = 0; i < TAMANHO; i++) {
32        printf("%d\t", vetor[i]);
33    }
34    printf("\n");
35    tempoFinal = clock(); //finaliza contagem do tempo
36    //calcula e mostra o tempo total de execucao
37    printf("Tempo: %f s\n", (double) (tempoFinal - tempoInicial) / CLOCKS_PER_SEC);
38 }
```

---

### Exercícios

1. Atualizar a implementação do *Insertion Sort* para executar as seguinte tarefa:
  - executar o algoritmo com três vetores de entrada:
    - (a) um vetor ordenado crescente de tamanho 100000 (cem mil)
    - (b) um vetor com números aleatórios de tamanho 100000 (cem mil)
    - (c) um vetor ordenado decrescente de tamanho 100000 (cem mil)
  - criar variáveis para contar a quantidade de comparações e quantidade de trocas;
2. Criar gráficos para as três execuções comparando quantidade de comparações e quantidade de trocas e o tempo de execução;

## Merge Sort

O algoritmo de ordenação *Merge Sort* é a técnica de divisão e conquista, ou seja, o problema é dividido em subproblemas menores até que a solução para o subproblema seja simples. As soluções dos subproblemas são combinadas para prover a solução do problema original.

No *Merge Sort*, o vetor é dividido em partes iguais recursivamente até que haja apenas um elemento. Por exemplo se um vetor tem  $n$  elementos, ele é dividido em  $n/2$ , o resultado é novamente dividido por 2 até que o vetor não seja mais divisível. Os pares da última divisão são ordenados entre si assim como na volta recursiva montando o vetor original que estará então ordenado. O programa 4.1 apresenta uma implementação em C do algoritmo *Merge Sort* para um vetor de tamanho 4.

Programa 4.4: Implementação em C do algoritmo de ordenação Merge Sort.

```

1 #include<stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4 #include <stdlib.h>
5 #define TAMANHO 4
6 void merge(int vetor[], int inicio, int meio, int fim);
7 void mergeSort(int vetor[], int inicio, int meio);
8
9 void main() {
10     int vetor[TAMANHO]; //vetor com tamanho definido
11     clock_t tempoInicial, tempoFinal; //Variaveis para guardar o tempo de execucao
12     srand(time(NULL)); //Cria uma semente para numeros aleatorios
13     tempoInicial = clock(); //inicia contagem do tempo
14     for (int i = 0; i < TAMANHO; i++) {
15         vetor[i] = rand() % 10; //Atribui um inteiro aleatorio entre 0 e 9
16     }
17     //Mostra valores do vetor nao ordenado
18     for (int i = 0; i < TAMANHO; i++) {
19         printf("%d\t", vetor[i]);
20     }
21     printf("\n");
22     //Chama a função passando o vetor como parametro
23     mergeSort(vetor, 0, TAMANHO - 1);
24     //Mostra valores do vetor ordenado
25     for (int i = 0; i < TAMANHO; i++) {
26         printf("%d\t", vetor[i]);
27     }
28     printf("\n");
29     tempoFinal = clock(); //finaliza contagem do tempo
30     //calcula e mostra o tempo total de execucao
31     printf("Tempo: %f s\n", (double) (tempoFinal - tempoInicial) / CLOCKS_PER_SEC);
32 }
33
34 void merge(int vetor[], int inicio, int meio, int fim) {
35     int i, j, k;
36     int n1 = meio - inicio + 1;
37     int n2 = fim - meio;
38     int vetorEsquerda[n1], vetorDireita[n2];
39     for (i = 0; i < n1; i++)
40         vetorEsquerda[i] = vetor[inicio + i];
41     for (j = 0; j < n2; j++)
42         vetorDireita[j] = vetor[meio + 1 + j];
43     i = 0;
44     j = 0;
45     k = inicio;
46     while (i < n1 && j < n2) {
47         if (vetorEsquerda[i] <= vetorDireita[j]) {
48             vetor[k] = vetorEsquerda[i];
49             i++;
50         }
51         else {//troca
52             vetor[k] = vetorDireita[j];
53             j++;
54         }
55     }
56 }
```

```
55         k++;
56     }
57     while (i < n1) {
58         vetor[k] = vetorEsquerda[i];
59         i++;
60         k++;
61     }
62
63     while (j < n2) {
64         vetor[k] = vetorDireita[j];
65         j++;
66         k++;
67     }
68 }
69
70 void mergeSort(int vetor[], int inicio, int fim) {
71     if (inicio < fim) { //condicao de parada
72         int m = inicio + (fim - inicio) / 2; //posicao para dividir o vetor
73         mergeSort(vetor, inicio, m); //chamada recursiva para a metade esquerda
74         mergeSort(vetor, m + 1, fim); //chamada recursiva para a metade direita
75         merge(vetor, inicio, m, fim);
76     }
77 }
```

---

## Exercícios

1. Atualizar a implementação do *Merge Sort* para executar as seguinte tarefa:
  - executar o algoritmo com três vetores de entrada:
    - (a) um vetor ordenado crescente de tamanho 100000 (cem mil)
    - (b) um vetor com números aleatórios de tamanho 100000 (cem mil)
    - (c) um vetor ordenado decrescente de tamanho 100000 (cem mil)
  - criar variáveis para contar a quantidade de comparações e quantidade de trocas;
2. Criar gráficos para as três execuções comparando quantidade de comparações e quantidade de trocas e o tempo de execução;

## Quick Sort

O algoritmo de ordenação *Quick Sort* também utiliza a técnica de divisão e conquista. No *Quick Sort*, o vetor é dividido em duas partes recursivamente a partir de um pivô escolhido, até que não seja possível dividir mais. Os pares da última divisão são ordenados entre si assim na volta recursiva colocando todos os valores menores que o pivô no vetor à sua esquerda e todos os valores maiores que o pivô no vetor à sua direita. Após esse procedimento o pivô estará ordenado em sua posição com relação os demais números. O programa 4.1 apresenta uma implementação em C do algoritmo *Quick Sort* para um vetor de tamanho 4.

Programa 4.5: Implementação em C do algoritmo de ordenação Quick Sort.

```

1 #include<stdlib.h>
2 #include <stdio.h>
3 #include <time.h>
4 #define TAMANHO 4
5 void trocar(int *x, int *y);
6 void quickSort(int vetor[], int inicio, int fim);
7
8 void main() {
9     int vetor[TAMANHO]; //vetor com tamanho definido
10    clock_t tempoInicial, tempoFinal; //Variaveis para guardar o tempo de execucao
11    srand(time(NULL)); //Cria uma semente para numeros aleatorios
12    tempoInicial = clock(); //inicia contagem do tempo
13    for (int i = 0; i < TAMANHO; i++) {
14        vetor[i] = rand() % 10; //Atribui um inteiro aleatorio entre 0 e 9
15    }
16    //Mostra valores do vetor nao ordenado
17    for (int i = 0; i < TAMANHO; i++) {
18        printf("%d\t", vetor[i]);
19    }
20    printf("\n");
21    //Chamada da funcao
22    quickSort(vetor, 0, TAMANHO - 1);
23    //Mostra valores do vetor ordenado
24    for (int i = 0; i < TAMANHO; i++) {
25        printf("%d\t", vetor[i]);
26    }
27    printf("\n");
28    tempoFinal = clock(); //finaliza contagem do tempo
29    //calcula e mostra o tempo total de execucao
30    printf("Tempo: %f s\n", (double) (tempoFinal - tempoInicial) / CLOCKS_PER_SEC);
31 }
32
33 void trocar(int *x, int *y) {
34     int aux; aux = *x; *x = *y; *y = aux;
35 }
36
37 void quickSort(int vetor[], int inicio, int fim) {
38     int pivo, i, j;
39     if(inicio < fim) { //condicao de parada
40         pivo = inicio;
41         i = inicio;
42         j = fim;
43         while (i < j) {
44             while(vetor[i] <= vetor[pivo] && i < fim)
45                 i++;
46             while(vetor[j] > vetor[pivo])
47                 j--;
48             if(i < j) {
49                 trocar(&vetor[i], &vetor[j]);
50             }
51         }
52         trocar(&vetor[pivo], &vetor[j]); //troca os valores usando ponteiros
53         quickSort(vetor, inicio, j - 1);
54         quickSort(vetor, j + 1, fim);
55     }
56 }
```

**Exercícios**

1. Atualizar a implementação do *Quick Sort* para executar as seguinte tarefa:
  - executar o algoritmo com três vetores de entrada:
    - (a) um vetor ordenado crescente de tamanho 100000 (cem mil)
    - (b) um vetor com números aleatórios de tamanho 100000 (cem mil)
    - (c) um vetor ordenado decrescente de tamanho 100000 (cem mil)
  - criar variáveis para contar a quantidade de comparações e quantidade de trocas;
2. Criar gráficos para as três execuções comparando quantidade de comparações e quantidade de trocas e o tempo de execução;

## 4.2 Estruturas de Dados Homogêneas e Heterogêneas

### Vetores

Primeiro, uma revisão sobre vetores. Em C uma variável que represente um vetor é um ponteiro. Portanto, quando um vetor é parâmetro para uma função o que é passado é a sua referência, ou seja, o endereço base do vetor. Vetores podem ser bi-, tri-, ou multi-dimensionais, mas abrigam o mesmo tipo de dados. O programa 4.2 mostra o vetor sendo passado para uma função que retorna o valor do meio do vetor.

Programa 4.6: Em C, vetores são passados para funções por referência.

---

```

1 #include <stdio.h>
2 #define TAMANHO 100
3 int getValorDoMeio(int *vetor, int tamanho);
4
5 void main(){
6     int vetor[TAMANHO];
7     int valorDoMeio;
8     for(int i=0; i<TAMANHO;i++){
9         vetor[i] = i+1;
10    }
11    valorDoMeio = getValorDoMeio(vetor,TAMANHO);
12    printf("Valor do meio: %d \n", valorDoMeio);
13 }
14
15 int getValorDoMeio(int *vetor, int tamanho){
16     printf("Valores no vetor:\n");
17     for(int i=0; i<tamanho;i++){
18         printf("%d\t",vetor[i]);
19     }
20     printf("\n");
21     return vetor[tamanho/2];
22 }
23 // Valores no vetor:
24 // 1   2   3   4   5   6   7   8   9   10
25 // 11  12  13  14  15  16  17  18  19  20
26 // 21  22  23  24  25  26  27  28  29  30
27 // 31  32  33  34  35  36  37  38  39  40
28 // 41  42  43  44  45  46  47  48  49  50
29 // 51  52  53  54  55  56  57  58  59  60
30 // 61  62  63  64  65  66  67  68  69  70
31 // 71  72  73  74  75  76  77  78  79  80
32 // 81  82  83  84  85  86  87  88  89  90
33 // 91  92  93  94  95  96  97  98  99  100
34 // Valor do meio: 51

```

---

Programa 4.7: Um vetor de duas dimensões mostrando apenas valores da diagonal principal.

---

```

1 #include <stdio.h>
2 #define TAMANHO 4
3 void main(){
4     int vetor[TAMANHO][TAMANHO];
5     for (int l=0; l<TAMANHO; l++){
6         for (int c=0; c<TAMANHO; c++){
7             printf("Valor [%d][%d]: ", l,c);
8             scanf("%d", &vetor[l][c]);
9         }
10    }
11
12    for (int l=0; l<TAMANHO; l++){
13        for (int c=0; c<TAMANHO; c++){
14            if(l == c){
15                printf("[%d][%d]:%d\t",l,c,vetor[l][c]);
16            }
17        }
18        printf("\n");
19    }
20 }

```

---

---

```

21 // Valor [0][0]: 1
22 // Valor [0][1]: 2
23 // Valor [0][2]: 2
24 // Valor [0][3]: 2
25 // Valor [1][0]: 2
26 // Valor [1][1]: 1
27 // Valor [1][2]: 2
28 // Valor [1][3]: 2
29 // Valor [2][0]: 2
30 // Valor [2][1]: 2
31 // Valor [2][2]: 1
32 // Valor [2][3]: 2
33 // Valor [3][0]: 2
34 // Valor [3][1]: 2
35 // Valor [3][2]: 2
36 // Valor [3][3]: 1
37 // [0][0]:1
38 // [1][1]:1
39 // [2][2]:1
40 // [3][3]:1

```

---

## Strings em C

Em C, uma *string* é um vetor de caracteres e cada *string* termina com um caractere *NULL*. Uma constante *string* é definida dentro de aspas em que o caractere *NULL* é automaticamente incluído. Por exemplo, a string "IFG" é um vetor de 4 elementos. O programa 4.2 mostra um exemplo de *string* em C.

Programa 4.8: Em C, *strings* são vetores de caracteres..

---

```

1 #include <stdio.h>
2 #define TAMANHO 100
3 void main(){
4     char vetor[TAMANHO] = "Aqui vai uma frase bastante longa para servir de exemplo!";
5     int qtd_de_letras_a_na_string = 0;
6     for (int i=0; i<TAMANHO; i++){
7         if (vetor[i] == 'a' || vetor[i] == 'A'){
8             qtd_de_letras_a_na_string++;
9         }
10    }
11    printf("Quantidade de letras '\\'a\' ou '\\'A\' na string: %d\n",
12           qtd_de_letras_a_na_string);
13 }

```

---

## Structs

Em C, uma *struct* é uma forma de criar novos tipos heterogêneos. O programa 4.2 mostra um exemplo de *struct* em C.

## Programa 4.9: Structs em C.

---

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <math.h>
4 #define CPF_TAM 14
5 #define NOME_TAM 100
6
7 typedef struct { // cria um novo tipo como struct
8     float peso; // campo peso
9     int idade; // campo idade
10    float altura; // campo altura
11    char cpf[CPF_TAM]; //campo cpf
12    char nome[NOME_TAM]; //campo nome
13 } Pessoa; // nome do novo tipo criado
14
15 /*Funcao para calcular o IMC*/
16 float calcularIMC(Pessoa p){
17     return p.peso / (p.altura*p.altura);
18 }
19
20 /*Funcao para imprimir dados da pessoa (parametro por valor)*/
21 void imprimirPessoa(Pessoa p) {
22     printf("CPF: %s\nNome: %s\nIdade: %d\nPeso: %.2f\nAltura: %.2f\n", p.cpf, p.nome,
23           p.idade, p.peso, p.altura);
24     printf("IMC: %.2f\n", calcularIMC(p));
25 }
26 /*Funcao para "preencher" uma pessoa (parametro por referencia)*/
27 void setPessoa(Pessoa* p, int idade, float peso, float altura, char cpf[], char nome
28                 []) {
29     // Quando usando ponteiros, o campo pode ser acessado de 2 formas:
30     // a) (*nome_do_ponteiro).nome_do_campo
31     // b) nome_do_ponteiro->nome_do_campo
32     (*p).idade = idade; //exemplo a)
33     p->peso = peso; //exemplo b)
34     p->altura = altura;
35     for (int i=0;i<CPF_TAM;i++)
36         p->cpf[i] = cpf[i];
37     for (int i=0;i<NOME_TAM;i++)
38         p->nome[i] = nome[i];
39 }
40
41 void main() {
42     Pessoa pessoa01;
43     char cpf[CPF_TAM] = "111.111.111-11";
44     char nome[NOME_TAM] = "Pessoa da Silva";
45     setPessoa(&pessoa01, 37, 70, 1.75, cpf, nome);
46     imprimirPessoa(pessoa01);
47 }
```

---

O programa 4.2 mostra um exemplo de composição de *struct* em C.

Programa 4.10: Composição de Structs em C.

---

```

1 #include <stdio.h>
2 #include <string.h>
3 #define TAM 100
4 typedef struct { // cria um novo tipo como struct
5     char nome[TAM]; //nome da marca
6     char nacionalidade[TAM]; //nacionalidade da marca
7 } Marca; // nome do novo tipo criado
8
9 typedef struct { // cria um novo tipo como struct
10    char modelo[TAM]; //modelo do carro
11    float motor; //motorizacao
12    Marca marca;
13 } Carro; // nome do novo tipo criado
14
15 void setMarca(Marca* marca, char nome[], char nacionalidade[]) {
16     for (int i=0;i<TAM;i++)
17         marca->nome[i] = nome[i];
18     for (int i=0;i<TAM;i++)
19         marca->nacionalidade[i] = nacionalidade[i];
20 }
21
22 void setCarro(Carro* carro, char modelo[], float motor, Marca marca) {
23     for (int i=0;i<TAM;i++)
24         carro->modelo[i] = modelo[i];
25     carro->motor = motor;
26     carro->marca = marca;
27 }
28
29 void printCarro(Carro carro){
30     printf("%s\n",carro.modelo);
31     printf("%.2f\n",carro.motor);
32     printf("%s\n",carro.marca.nacionalidade);
33     printf("%s\n",carro.marca.nome);
34 }
35
36 void main(){
37     Marca ford;
38     Marca vw;
39     setMarca(&ford, "Ford", "EUA");
40     setMarca(&vw, "VW", "Alemanha");
41     Carro gol;
42     setCarro(&gol, "Gol", 1.0, vw);
43     printCarro(gol);
44 }
```

---

### 4.3 Listas Encadeadas

Uma lista encadeada é uma estrutura de dados que permite um crescimento sob demanda, limitado à capacidade de memória disponível no computador. Ela é composta de “nós” que tem suas posições de memória ligadas através de ponteiros. Cada nó da lista, além de guardar dados em si, aponta o próximo nó. Os dados em um nó também podem ser heterogêneos. A Figura 4.2 ilustra essa ideia. Apesar da figura dar a impressão de que os nós estão distribuídos contiguamente na memória, eles estão na verdade espalhados em endereços aleatórios de memória. Por isso o encadeamento funciona com ponteiros apontando o endereço de memória do próximo nó da lista. O programa 4.3 apresenta uma implementação em C de uma lista encadeada.

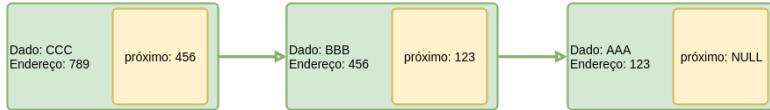


Fig. 4.2: Representação de um Lista Encadeada.

Programa 4.11: Implementação em C de lista encadeada.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 // Define o tipo No contendo
6 typedef struct No{
7     int dado;
8     struct No *link;
9 } No;
10
11 void printFormat01(No* no);
12 void printFormat02(No* no);
13 //Define o ultimo No da lista
14 No* cabeca = NULL;
15
16 //Funcao que adiciona dados
17 void inserir(int dado) {
18     No* no;
19     no = (No *) malloc(sizeof(No));
20     no->dado = dado;
21     no->link = NULL;
22     if (cabeca == NULL)
23         cabeca = no;
24     else {
25         no->link = cabeca;
26         cabeca = no;
27     }
28 }
29
30
31 //Funcao que imprime a lista
32 void imprimirLista() {
33     No* no;
34     if (cabeca == NULL) {
35         printf("Lista vazia.\n");
36         return;
37     }
38     no = cabeca;
39     while (no != NULL) {
40         if (no->link != NULL){
41             printFormat01(no);
42         }else{
43             printFormat02(no);
44         }
45         no = no->link;
46     }
47 }
48 }
49
50 void printFormat01(No* no){
51     printf("[%d(%p)|%p]\n", no->dado, no, no->link);
52     printf("          |\\n");
53     printf("          V\\n");
54     printf("          -----\\n");
55     printf("          |\\n");
56     printf("          V\\n");
57 }
58 void printFormat02(No* no){
59     printf("[%d(%p)|%p]\n", no->dado, no, no->link);
60     printf("          |\\n");
  
```

```
61     printf("          V\n");
62     printf("          NULL\n");
63 }
64
65 void buscarDado(int dado) {
66     No* no;
67     if (cabeca == NULL) {
68         printf("Lista vazia.\n");
69         return;
70     }
71     no = cabeca;
72     while (no != NULL) {
73         if (no->dado == dado)
74             printf("[%d(%p)]\n", no->dado, no);
75         no = no->link;
76     }
77 }
78
79 void removerDado(int dado) {
80     No *no, *anterior;
81     if (cabeca == NULL) {// lista vazia
82         return;
83     } else { // lista NAO vazia
84         no = cabeca;
85         anterior = cabeca;
86         while (no != NULL) {
87             if (no->dado == dado){
88                 if (no == cabeca){// removendo o primeiro
89                     cabeca = cabeca->link;
90                     free(no); // libera memoria
91                     return;
92                 }
93                 else{ // removendo do meio
94                     anterior->link = no->link;//refaz links
95                     free(no); // libera memoria
96                     return;
97                 }
98             }
99             else{ // continua procurando na lista
100                 anterior = no;
101                 no = no->link;
102             }
103         }
104     }
105 }
106
107
108 void main() {
109     // Insere na lista os numeros de 1 a 3
110     for (int i = 1; i <= 4; i++)
111         inserir(i);
112     imprimirLista();
113     removerDado(2);
114     printf("-----\n");
115     imprimirLista();
116     buscarDado(3);
117 }
```

As listas encadeadas podem conter ponteiros para outro nós além do próximo. É possível criar uma lista com *links* simultâneos para o próximo nó e para o nó anterior. Essas listas são conhecidas como listas duplamente encadeadas. O programa 4.3 apresenta uma implementação em C de uma duplamente encadeada.

Programa 4.12: Implementação em C de lista duplamente encadeada.

---

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 // Define o tipo No contendo
6 typedef struct No{
7     struct No *proximo;
8     struct No *anterior;
9     int dado;
10 } No;
11
12 //Define o ultimo No da lista
13 No* cabeca = NULL;
14
15 //Funcao que adiciona dados
16 void inserir(int dado) {
17     No* no = (No *) malloc(sizeof (no));
18     no->dado = dado;
19     if (cabeca == NULL) {//lista vazia
20         cabeca = no;
21         no->proximo = NULL;
22         no->anterior = NULL;
23     }
24     else {
25         no->proximo = cabeca;
26         cabeca->anterior = no;
27         no->anterior = NULL;
28         cabeca = no;
29     }
30 }
31
32 //Funcao que imprime a lista
33 void imprimirLista() {
34     if (cabeca == NULL) {
35         printf("Lista vazia.\n");
36         return;
37     }
38     No* no = cabeca;
39     while (no != NULL) {
40         printf("|%p|%d(%p)|%p|\n", no->anterior, no->dado, no, no->proximo);
41         no = no->proximo;
42     }
43     printf("\n");
44 }
45
46 void main() {
47     // Insere na lista os numeros de 1 a 3
48     for (int i = 1; i <= 5; i++)
49         inserir(i);
50     imprimirLista();
51 }
```

---

#### 4.4 Pilhas

#### 4.5 Filas

## 4.6 Árvores

O programa 4.6 apresenta uma implementação em C de uma árvore binária. A representação gráfica da árvore é feita usando a biblioteca [Graphviz](#) (ELLSON et al., 2003).

Programa 4.13: Implementação em C de lista duplamente encadeada.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 typedef struct No{
6     int dado;
7     struct No* direita;
8     struct No* esquerda;
9 } No;
10
11 No* criarArvore(){ return NULL; }
12
13 int ArvoreVazia(No* raiz){ // 1 se a arvore vazia, 0 caso contrario
14     return raiz == NULL;
15 }
16
17 int getValor(No** no){
18     if ((*no) != NULL){
19         return (*no)->dado;
20     }
21 }
22
23 void mostrarArvore(No* raiz){
24     if (!ArvoreVazia(raiz)){ //No nao vazio
25         printf("%p<-%d(%p)->%p\n\n", raiz->esquerda, raiz->dado, raiz, raiz->direita);
26         printf("%d", getValor(&raiz->esquerda));
27         mostrarArvore(raiz->esquerda); //esquerda (subNo)
28         mostrarArvore(raiz->direita); //direita (subNo)
29     }
30 }
31
32 // Metodo para desenhar a arvore em um arquivo arvore.png
33 // Precisa de ter o graphviz instalado
34 // Ubuntu: sudo apt install python-pydot python-pydot-ng graphviz
35 // Fedora: sudo dnf install graphviz
36 void gerarArquivoDot(FILE** arquivoDot, No* raiz){
37     if (raiz != NULL){
38         char s1[20];
39         char s2[20];
40         if (getValor(&raiz->esquerda)!=0){
41             sprintf(s1, "%d:sw->%d [ label=%\"esq\"\"];\\n", raiz->dado, getValor(&raiz->esquerda));
42             fprintf((*arquivoDot), "%s", s1);
43         }
44         if (getValor(&raiz->direita)!=0){
45             sprintf(s2, "%d:se->%d [ label=%\"dir\"\"];\\n", raiz->dado, getValor(&raiz->direita));
46             fprintf((*arquivoDot), "%s", s2);
47         }
48         gerarArquivoDot(arquivoDot, raiz->esquerda); //esquerda (subNo)
49         gerarArquivoDot(arquivoDot, raiz->direita); //direita (subNo)
50     }
51 }
52
53 void buscarDado(No** raiz, int dado){
54     if (!ArvoreVazia(*raiz)){ //No nao vazio
55         if (dado == (*raiz)->dado){
56             printf("%d encontrado.\n", dado);
57             return;
58         }
59     else{
56         if (dado < (*raiz)->dado){ //dado menor? vai pra esquerda
57             buscarDado(&(*raiz)->esquerda, dado);
58         }
59     }
60 }
61

```

```

62         }
63         if(dado > (*raiz)->dado){ //dado maior? vai pra direita
64             buscarDado(&(*raiz)->direita, dado);
65         }
66     }
67 }
68 }
69
70 void inserirDado(No** raiz, int dado){
71     if(*raiz == NULL){
72         *raiz = (No*)malloc(sizeof(No));
73         (*raiz)->esquerda = NULL;
74         (*raiz)->direita = NULL;
75         (*raiz)->dado = dado;
76     } else {
77         if(dado < (*raiz)->dado){ //dado menor? vai pra esquerda
78             //percorrer subNo da esquerda
79             inserirDado(&(*raiz)->esquerda, dado);
80         }
81         if(dado > (*raiz)->dado){ //dado maior? vai pra direita
82             //percorrer subNo da direita
83             inserirDado(&(*raiz)->direita, dado);
84         }
85     }
86 }
87
88 int getAltura(No* raiz) {
89     if (raiz == NULL) return -1;
90     else {
91         int hEsquerda = getAltura(raiz->esquerda);
92         int hDireita = getAltura(raiz->direita);
93         return (hEsquerda < hDireita) ? hDireita + 1 : hEsquerda + 1;
94     }
95 }
96
97 void main(){
98     No* raiz = criarArvore();
99     srand(time(0));
100    for (int i = 0; i < 50; i++) {
101        inserirDado(&raiz, rand() % 100);
102    }
103    buscarDado(&raiz, 7);
104    printf("Altura: %d\n", getAltura(raiz));
105    FILE* arquivoDot;
106    arquivoDot = fopen("arvore.dot", "w");
107    fprintf(arquivoDot, "%s", "digraph G {\nsplines=line;\n");
108    gerarArquivoDot(&arquivoDot, raiz);
109    fprintf(arquivoDot, "%s", "}\n");
110    fclose(arquivoDot);
111    free(raiz);
112    system("dot -Tpng arvore.dot -o arvore.png");
113 }

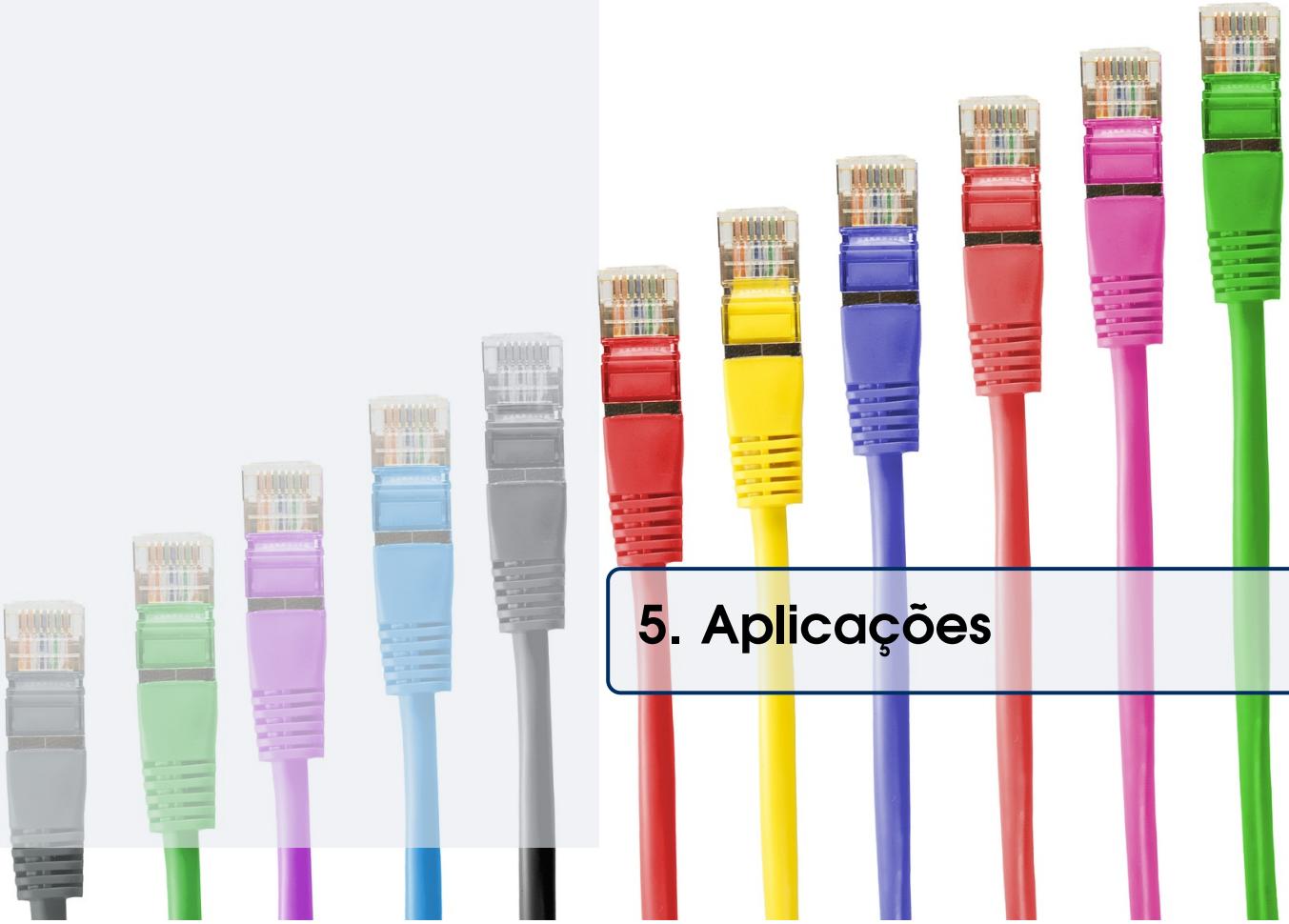
```

---

## 4.7 Tabelas Hashing

## 4.8 Grafos

### 4.8.1 Busca em Grafos



## 5. Aplicações

## Referências Bibliográficas

ASCENCIO, Ana Fernanda G.; ARAÚJO, Graziela S. de. Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, v. 3, 2010.

CORMEN, Thomas H. et al. **Introduction to algorithms**. [S.l.]: MIT press, 2009.

ELLSON, John et al. Graphviz and dynagraph – static and dynamic graph drawing tools. In: GRAPH DRAWING SOFTWARE. [S.l.]: Springer-Verlag, 2003. p. 127–148.

# 6. Apêncice