

**INSTITUTO
FEDERAL**

Goiás

Câmpus
Formosa

Algoritmos e Estruturas de Dados

Tecnólogo em Análise e Desenvolvimento de Sistemas



Waldeyr Mendes Cordeiro da Silva

2019

Material didático para Algoritmos e Estruturas de Dados

Versão 0.1

Waldeyr Mendes Cordeiro da Silva

- Formação:
 - 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>

Sumário

1	Prefácio	5
2	Introdução	6
2.1	Análise de Algoritmos	7
2.1.1	Elementos de Notação Assintótica	7
3	Programação	8
3.1	Primeiros Passos em Programação	9
4	Estruturas de Dados	10
4.1	Algoritmos de Ordenação	10
4.2	Estruturas de Dados Homogêneas e Heterogêneas	19
4.3	Listas Encadeadas	22
4.4	Listas	24
4.5	Pilhas	24
4.6	Filas	24
4.7	Tabelas Hashing	24
4.8	Árvores	24
4.9	Grafos	24
4.9.1	Busca em Grafos	24
5	Aplicações	25

6	Apêndice	27
----------	-----------------------	-----------

A globe is centered in the upper half of the page, overlaid with a white network of dots and lines. The background is a deep space scene with a dense field of stars and a bright, hazy nebula on the left. A semi-transparent dark blue box with a thin white border is positioned in the lower right of the image area, containing the chapter title.

1. Prefácio

Em construção...

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 em aberto e cujo prêmio é de 1 milhão de dólares por cada solução.

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 significativo.

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, consituem 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).

2.1 Análise de Algoritmos

2.1.1 Elementos de Notação Assintótica

3. Programação

3.1 Primeiros Passos em Programação

Programa 3.1: Meu primeiro programa em C.

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

4. Estruturas de Dados

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.

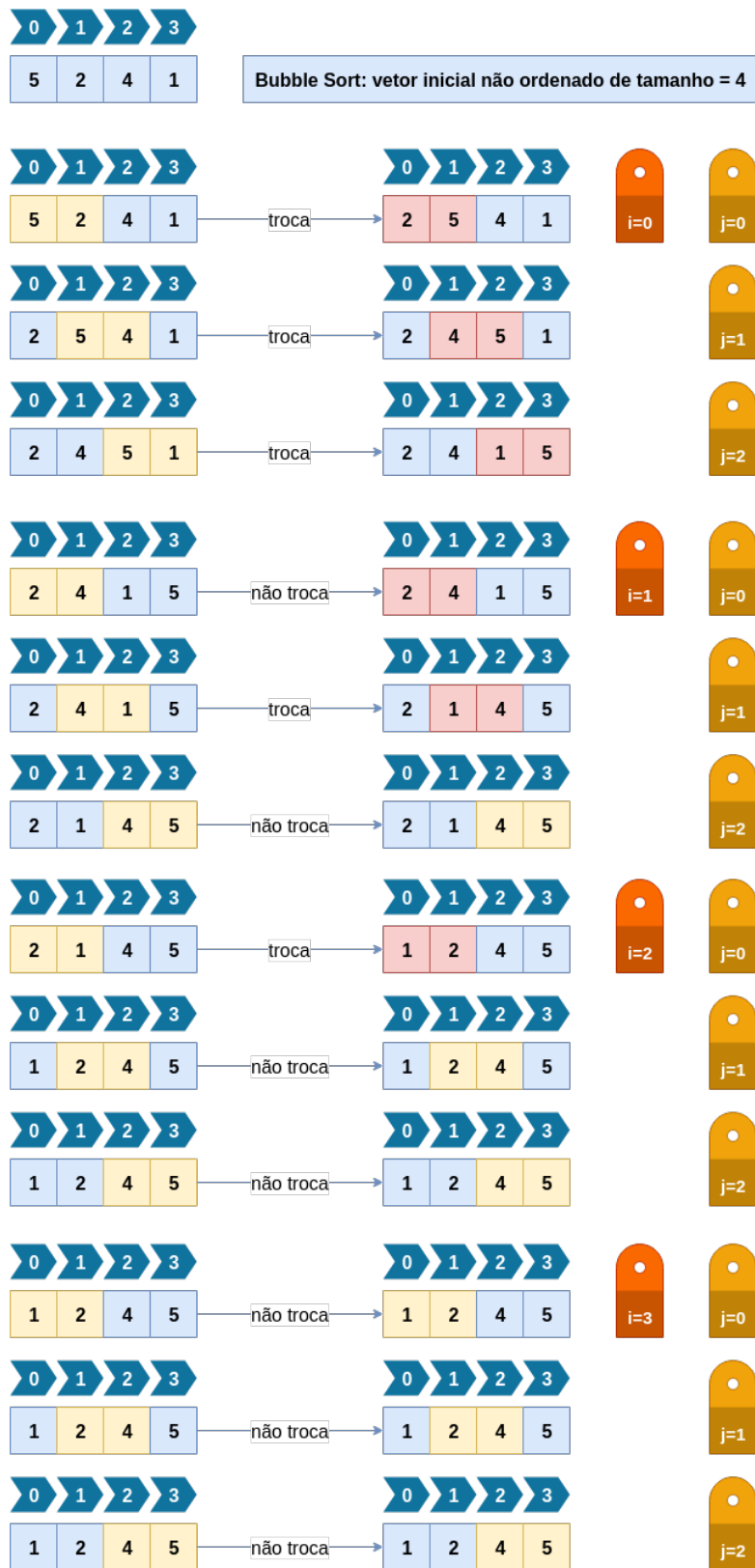
Programa 4.1: Implementação em C do algoritmo de ordenação BubbleSort.

```
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 número¹ 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

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

¹Ou 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 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

- Atualizar a implementação do *Insertion Sort* para executar as seguinte tarefa:
 - executar o algoritmo com três vetores de entrada:
 - um vetor ordenado crescente de tamanho 100000 (cem mil)
 - um vetor com números aleatórios de tamanho 100000 (cem mil)
 - um vetor ordenado decrescente de tamanho 100000 (cem mil)
 - criar variáveis para contar a quantidade de comparações e quantidade de trocas;
- 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* 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 fucao 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     }

```

```
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 caracter *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 }
14 // Quantidade de letras \'a\' ou \'A\' na string: 8

```

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.

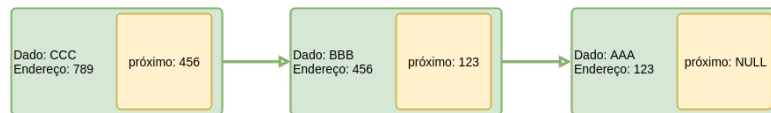


Fig. 4.2: Representação de um Lista Encadeada.

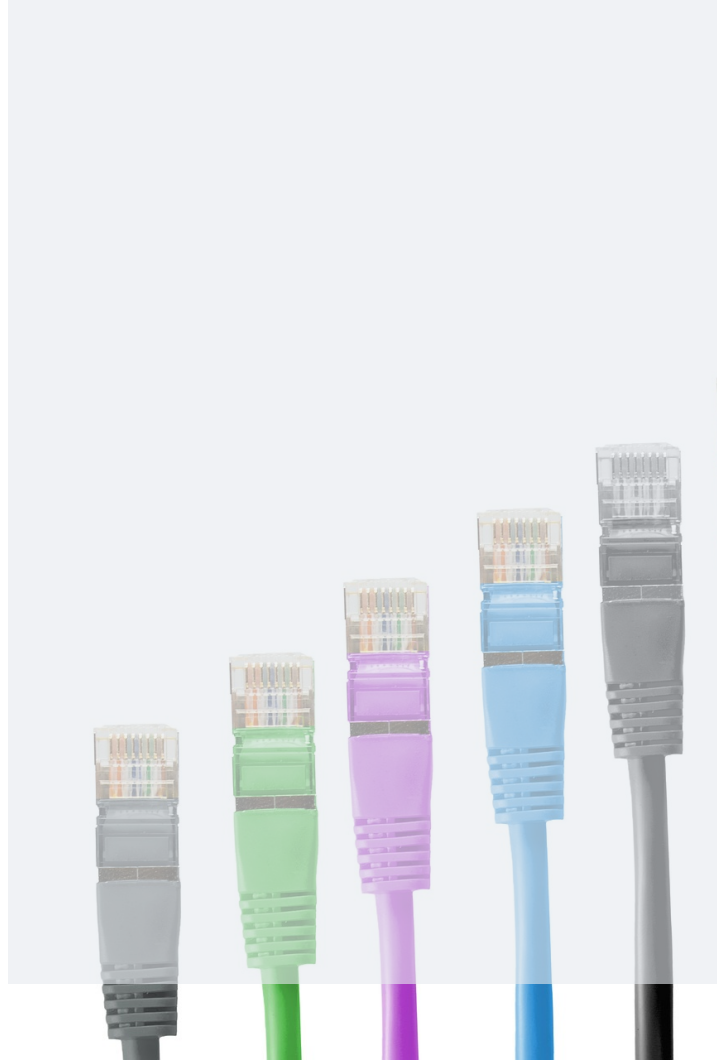
Programa 4.11: Implementação em C de lista encadeada.

```

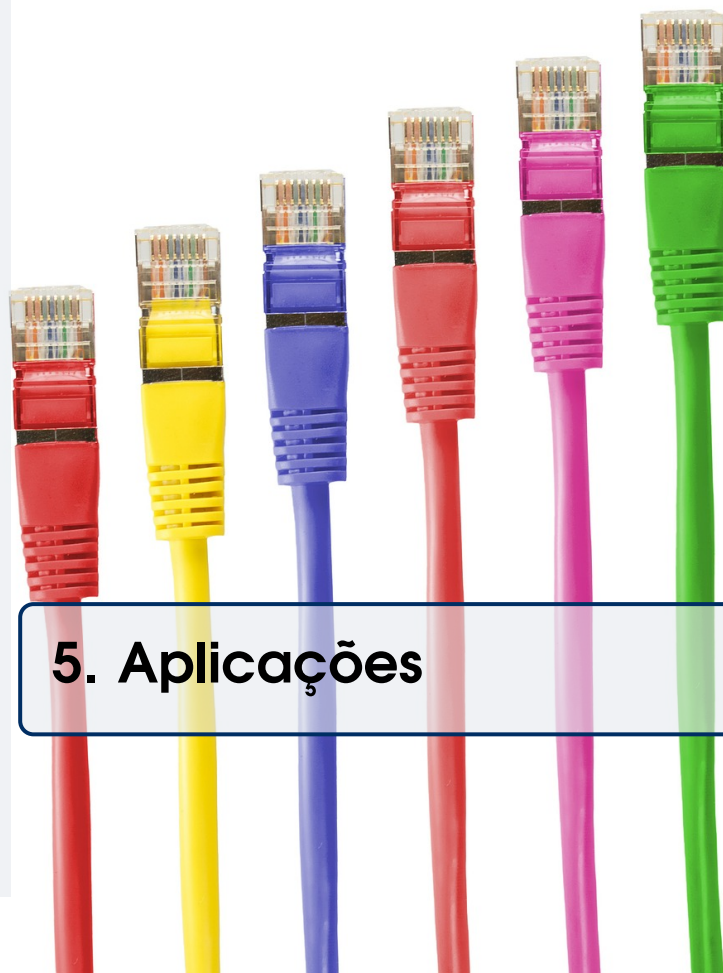
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  // Define o tipo No contendo
6  typedef struct No{
7      int dado;
8      struct No *prox;
9  } No;
10
11 //Define o ultimo No da lista
12 No* ponteiroFinal = NULL;
13
14 // Funcao que define a lista como vazia.
15 void criarLista() {
16     ponteiroFinal = NULL;
17 }
18
19 //Funcao que adiciona dados
20 void adicionarDado(int dado) {
21     No* ponteiroNo;
22     ponteiroNo = (No *) malloc(sizeof (ponteiroNo));
23     ponteiroNo->dado = dado;
24     ponteiroNo->prox = NULL;
25     if (ponteiroFinal == NULL)
26         ponteiroFinal = ponteiroNo;
27     else {
28         ponteiroNo->prox = ponteiroFinal;
29         ponteiroFinal = ponteiroNo;
30     }
31 }
32
33 //Funcao que imprime a lista
34 void imprimirLista() {
35     printf("-----\n");
36     No* ponteiroNo;
37     if (ponteiroFinal == NULL) {
38         printf("Lista vazia.\n");
39         return;
40     }
41     ponteiroNo = ponteiroFinal;
42     while (ponteiroNo != NULL) {
43         printf("[%d(%p) | %p]\n", ponteiroNo->dado, ponteiroNo, ponteiroNo->prox);
44         ponteiroNo = ponteiroNo->prox;
45     }
46     printf("-----\n");
47 }
48
49 void main() {
50     criarLista();
51     // Insere na lista os numeros de 1 a 5
52     for (int i = 1; i <= 5; i++)
53         adicionarDado(i);
54     imprimirLista();
55 }

```

- 4.4** Listas
- 4.5** Pilhas
- 4.6** Filas
- 4.7** Tabelas Hashing
- 4.8** Árvores
- 4.9** Grafos
 - 4.9.1** 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: Perarson Prentice Halt**, v. 3, 2010.

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

6. Apêndice