

ESTRUTURA DE DADOS

Patricia Rucker de Bassi



Estrutura de Dados

Patricia Rucker de Bassi



Curitiba
2017

Ficha Catalográfica elaborada pela Fael. Bibliotecária – Cassiana Souza CRB9/1501

B321e Bassi, Patricia Rucker de

Estrutura de dados / Patricia Rucker de Bassi. – Curitiba: Fael, 2017.

200 p.; il.

ISBN 978-85-60531-95-0

1. Estrutura de dados (Computação) I. Título

CDD 005.73

Direitos desta edição reservados à Fael.

É proibida a reprodução total ou parcial desta obra sem autorização expressa da Fael.

FAEL

Direção Acadêmica Francisco Carlos Sardo

Coordenação Editorial Raquel Andrade Lorenz

Revisão Editora Coletânea

Projeto Gráfico Sandro Niemicz

Capa Vitor Bernardo Backes Lopes

Imagen da Capa Shutterstock.com/Gajus

Arte-Final Evelyn Caroline dos Santos Betim

Sumário

CARTA AO ALUNO | 5

1. ESTRUTURA DE DADOS HOMOGÊNEOS E HETEROLOGÊNEOS | 7
2. ESTRUTURAS DE DADOS ELEMENTARES: PILHA | 23
3. ESTRUTURA DE DADOS ELEMENTARES: FILA | 35
4. PONTEIROS E ALOCAÇÃO DINÂMICA | 49
5. ESTRUTURA DE DADOS LISTA | 61
6. RECURSIVIDADE | 81
7. ÁRVORES | 95
8. GRAFOS | 113
9. MÉTODOS DE ORDENAÇÃO E PESQUISA | 129
10. COMPLEXIDADE DE ALGORITMOS | 143

CONCLUSÃO | 159

GABARITO | 161

REFERÊNCIAS | 195

Carta ao Aluno

PREZADO(A) ALUNO(A),

O CONTEÚDO ABORDADO nesta disciplina contribuirá para o aperfeiçoamento de sua técnica de programação. A estrutura de dados tem como objetivo organizar dados na memória do computador. Para isso, técnicas e algoritmo são apresentados e discutidos ao longo dos capítulos. O foco desta disciplina está na implementação computacional, porém não foca em uma linguagem de programação específica, já que apresenta algoritmos de solução dos problemas que podem ser traduzidos para a linguagem desejada. Os algoritmos estão em forma de funções ou procedimento, utilizam a programação estruturada e apresentam comentários para melhor entendimento da tarefa que está sendo executada.

Estrutura de Dados

Desta forma, é importante que você já conheça algoritmos, pois seu entendimento será necessário para compreender melhor o conteúdo dos capítulos.

Programas das mais diversas áreas fazem uso de técnicas de estrutura de dados, para tornar o armazenamento e o acesso a memória mais eficientes. Por isso essa disciplina é tão importante.

O conteúdo está apresentado nos capítulos de forma objetiva e simples, com exemplos que podem ser aplicados no seu cotidiano para facilitar o entendimento e para mostrar como a organização de dados é usual.

Esse livro não tem a pretensão de esgotar o estudo de estrutura de dados, pois algoritmos mais complexos podem e devem ser estudados. Mas, espero que você use este livro como uma ferramenta. Que ele possa contribuir para sua evolução profissional, revendo conceitos, adquirindo novos conhecimentos e testando situações que podem ser implantadas em cenários reais.

Desejo um bom aproveitamento dos conteúdos!

1

Estrutura de Dados Homogêneos e Heterogêneos

A CARACTERÍSTICA PRINCIPAL das estruturas de dados homogêneos e heterogêneos é a criação de muitas variáveis de mesmo tipo de dado (homogêneos), ou tipos de dados diferentes (heterogêneos), que atenderão pelo mesmo nome e que serão acessadas pelo seu deslocamento dentro da estrutura (homogêneos) ou por seus campos (heterogêneos). Esta situação simplifica consideravelmente a escrita de algoritmos, pois o computador continua alocando o mesmo espaço de memória.

VEJA ESTA SITUAÇÃO: criar um algoritmo que leia a nota de dez alunos, calcule a média destas notas e imprima o número de alunos com nota abaixo da média, na média e acima da média.

PARA SOLUCIONAR ESSE exercício, necessita-se criar dez variáveis de nota, sendo uma para cada aluno, uma variável soma e uma variável média. Não é nada exagerado, porém se for preciso realizar a mesma tarefa para uma turma de 120 alunos, o processo fica complicado em relação à definição de variáveis.

No caso citado, fica claro que os tipos de dados primitivos (inteiro, real, caractere e lógico) não são suficientes para representar toda e qualquer informação que possa surgir. Portanto, em muitas situações necessita-se “construir novos tipos de dados” a partir da composição de tipos primitivos (KANTEK; DE BASSI, 2013).

Objetivos de aprendizagem:

1. conhecer estruturas de dados homogêneos e heterogêneos;
2. conhecer a sintaxe de declaração de novas estruturas de dados nos algoritmos;
3. aplicar estas novas estruturas de dados na solução de problemas.

1.1 Novos tipos de dados

Estes “novos tipos de dados” têm um formato denominado *estrutura de dados*, que define como os tipos primitivos de dados estão organizados. Fazendo a comparação da memória do computador com um armário de gavetas, em que cada posição da memória seria uma gaveta, em um primeiro momento uma gaveta poderia comportar apenas uma informação e, segundo esse novo conceito, uma gaveta pode comportar um conjunto de informações primitivas, desde que devidamente organizadas. Estes novos tipos são estranhos ao algoritmo, logo devem ser declarados detalhes de sua estrutura (KANTEK; DE BASSI, 2013).

Desta forma, assim como na *teoria dos conjuntos*, uma variável pode ser interpretada como um elemento e uma *estrutura de dados* como um conjunto. Quando uma determinada estrutura de dados for composta de variáveis com o mesmo tipo primitivo, teremos um conjunto homogêneo de dados; porém, caso os elementos do conjunto não sejam do mesmo tipo primitivo de dados, então diz-se um conjunto heterogêneo de dados (KANTEK; DE BASSI, 2013).

Os novos tipos de dados podem ser classificados da seguinte forma:

- × tipos de dados homogêneos unidimensionais – vetores;
- × tipos de dados homogêneos multidimensionais – matrizes;

- ✗ tipos de dados heterogêneos – registros.

A utilização deste tipo de estrutura de dados recebe diversos nomes, como: variáveis indexadas, variáveis compostas, variáveis subscriptas, arranjos, vetores, matrizes, tabelas em memória ou *array* (termo em inglês para arranjo). No contexto deste livro, utilizaremos os nomes: vetores, matrizes e registros (KANTEK; DE BASSI, 2013).

1.2 Vetores

Vetores são estruturas de dados homogêneos unidimensionais, e sua utilização mais comum está vinculada à criação de tabelas. Este novo tipo de dado permite a criação de muitas variáveis de mesmo tipo de dado, que atenderão pelo mesmo nome, e que serão acessadas pelo deslocamento dentro do vetor, os índices (KANTEK; DE BASSI, 2013).

Para entender variáveis compostas unidimensionais, imagina-se um edifício com um número finito de andares, representando uma estrutura de dados, e seus andares, partições desta estrutura. Visto que os andares são uma segmentação direta do prédio, estes compõem então a chamada estrutura unidimensional, isto é, de uma só dimensão. (KANTEK; DE BASSI, 2013).

A declaração de vetores em algoritmos dá-se em duas etapas:

- ✗ **primeiro** devemos definir um novo tipo de dado que virá a se juntar aos já existentes (inteiro, real, caractere e lógico), e que vigorará dentro deste bloco de programa.

```
TIPO <nome do tipo> = VETOR [ lim inf : lim sup] <tipo de
dado já existente>;
```

Onde: lim inf – limite inferior do vetor;

lim sup – limite superior do vetor.

- ✗ **segundo**, devemos informar ao algoritmo quais variáveis poderão conter este novo tipo de dado neste bloco de programa.

```
<identificador 1>, ... , <identificador n> : <nome do novo
tipo>;
```

O número de elementos de um vetor será dado por:

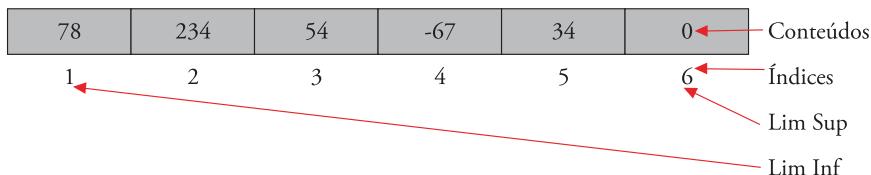
Estrutura de Dados

$$\text{Lim Sup} - \text{Lim Inf} + 1$$

Isto significa que as posições do vetor são identificadas a partir de Lim Inf, com incrementos unitários até Lim Sup (KANTEK; DE BASSI, 2013.)

A figura 1.1 mostra a representação de uma estrutura vetor com seus conteúdos, índices (posições), limite inferior do vetor e limite superior do vetor.

Figura 1.1 – Representação de um vetor



Fonte: Elaborada pelo autor.

Exemplo 1.1

1. Definir um vetor para conter as taxas inflacionárias dos 12 últimos meses:

```
TIPO VETINF = VETOR [ 1: 12] real;  
INFLACAO : VETINF;
```

2. Definir um vetor para conter os nomes de 60 alunos:

```
TIPO VET1 = VETOR [ 1: 60] caractere[50];  
NOME : VET1;
```

1.2.1 Acesso aos dados do vetor

Ao imaginar um elevador de um prédio, sabe-se que este é capaz de acessar qualquer um de seus andares. Entretanto, não basta saber que andar desejamos atingir se não soubermos o nome do edifício. Logo, necessita-se ter conhecimento do nome do edifício para então procurar o andar desejado. O mesmo acontece com os vetores, visto que são compostos por diversas variáveis. Torna-se então necessário definir o nome do vetor que contém os dados desejados e depois especificar em qual posição a informação se encontra (KANTEK; DE BASSI, 2013).

Após isolar um único elemento do vetor, pode-se manipulá-lo por meio de qualquer operação de entrada, saída ou atribuição. O acesso aos dados é feito de maneira individual, pressupondo-se a existência de um comando para cada elemento do vetor (KANTEK; DE BASSI, 2013).

```
Tipo VET = VETOR [ 1: 5] inteiro;
V1: VET;
V1[1] ← 2;
V1[2] ← 5;
V1[3] ← 6;
V1[4] ← 9;
V1[5] ← 23;
```

Outra forma de acesso às posições do vetor é utilizando um comando de repetição. Neste caso, o mais adequado é o comando **Para** (KANTEK; DE BASSI, 2013).

```
Tipo VET = VETOR [ 1: 5] inteiro;
V1: VET;
I: inteiro;
Para I de 1 até 5 faça
    Leia (V1[I]);
Fimpara;
```

Pode-se observar neste exemplo que o nome (V1) é um só, o que muda é a informação indicada dentro dos colchetes (1 até 5). Este valor é o índice que indica o endereço onde o elemento está armazenado. É necessário salientar a diferença entre o **conteúdo** do vetor (2, 5, 6, 9, e 23) e seu endereço ou **posição**, que é representado pelo índice (KANTEK; DE BASSI, 2013).

Esta forma de acesso ao vetor aplica-se à inicialização do vetor, isto é, podem ser utilizados pelos comandos de atribuição (\leftarrow ou $:=$) e entrada de dados (leia). Também podemos utilizar esta sintaxe no caso de saída de dados, isto é, o comando imprima ou escreva (KANTEK; DE BASSI, 2013).

Exemplo 1.2

Definir um algoritmo que leia um vetor de 120 elementos inteiros e imprima o somatório de seus elementos (**conteúdo**) e os elementos.

Estrutura de Dados

```
inicio
    Tipo VET = VETOR [1:120] inteiro;
    V : VET;
    I, SOMA: inteiro;
    SOMA ← 0;
    Para I de 1 até 120 faça
        Leia (V[I]);
        SOMA ← SOMA + V[I]; // conteúdo do vetor
    fimpara;
    imprima (SOMA);
    Para I de 1 até 120 faça
        imprima (V[I]);
    fimpara;
fim.
```

Fonte: Kantek; De Bassi (2013).

Exemplo 1.3

Escrever um algoritmo que leia um vetor contendo 1050 nomes de candidatos à vaga de emprego e imprima os nomes das **posições ímpares**.

```
inicio
    Tipo VET = VETOR [1:1050] caractere[50];
    V : VET;
    I: inteiro;
    Para I de 1 até 1050 faça
        Leia (V[I]);
    fimpara;
    Para I de 1 até 1050 passo 2 faça //somente as posições ímpares do vetor
        imprima(V[I]);
    fimpara;
fim.
```

Fonte: Kantek; De Bassi (2013).

Exemplo 1.4

Escrever um algoritmo que defina um vetor com 12 elementos lógicos e inicialize as 6 primeiras posições com o valor verdadeiro e as outras 6 com o valor falso.

```
inicio
    Tipo VET = VETOR [1:12] lógico;
```

```

V : VET;
I: inteiro;
Para I de 1 até 12 faça
    Se I ← 6    // posição
        Então V[I] ← V; // conteúdo
        Senão V[I] ← F; // conteúdo
    fimse;
fimpara;
Para I de 1 até 12 faça
    imprima(V[I]);
fimpara;
fim.

```

Fonte: Kantek; De Bassi (2013).

Exemplo 1.5

Escrever um algoritmo que leia dois vetores A e B com 20 elementos cada e construir um vetor C, onde cada elemento de C é a subtração do elemento correspondente de A e B.

```

inicio
    Tipo VET = VETOR [1:20] real;
    A, B, C: VET;
    I: inteiro;
    Para I de 1 até 20 faça
        Leia (A[I]);
        Leia (B[I]);
        C[I] ← A[I] - B[I];
    fimpara;
    Para I de 1 até 20 faça
        imprima(C[I]);
    fimpara;
fim.

```

Fonte: Kantek; De Bassi (2013).



Saiba mais

Algumas linguagens de programação, como a linguagem C, inicializam automaticamente o índice da estrutura homogênea com o valor 0. Desta forma, um vetor de 10 posições em C terá os índices de 0 até 9. Nestes casos, não é preciso indicar o valor inicial do índice.

da estrutura homogênea, somente quantas posições a estrutura terá.
Observe o exemplo a seguir.

Declarar um vetor de 10 posições de conteúdo real em C:

```
VET [10] float;
```



1.3 Matrizes

Pelas mesmas razões que foi criado o conceito de vetor, necessita-se tratar de outro tipo de organização de dados: as matrizes. As estruturas de dados vetores são variáveis indexadas com apenas uma dimensão, isto é, uma coluna e várias linhas. A partir de agora serão apresentadas tabelas com mais colunas, sendo assim, haverá variáveis no sentido horizontal e vertical. As mais comuns são as matrizes de duas dimensões, por se relacionarem diretamente com a utilização de tabelas. Matrizes com mais de duas dimensões são utilizadas com menos frequência, mas poderão ocorrer momentos em que se necessite trabalhar com um número maior de dimensões (KANTEK; DE BASSI, 2013).

No caso de matriz bidimensional, necessitaremos de dois índices para referenciar um determinado elemento: linha e coluna. Para uma matriz tridimensional, usaremos três índices: plano, linha e coluna. Da mesma forma que a manipulação de um vetor necessita de uma instrução de repetição (enquanto, repita ou para), no caso de matrizes deverá ser utilizado o número de repetições relativo ao número de dimensões da matriz. Sendo assim, uma matriz de duas dimensões deverá ser controlada por duas repetições, uma de três dimensões fará uso de três repetições e assim por diante (KANTEK; DE BASSI, 2013).

Para definirmos uma matriz passamos por duas etapas:

- × **1^a etapa** – definição do tipo de dado

```
Tipo <nome do tipo> = Matriz [li1:ls1,li2:ls2] <tipo  
de dados já existente>;
```

Onde: li1, li2, liN – são os limites inferiores da primeira, segunda e enésima dimensão da matriz, respectivamente.

Ls1, ls2, lsN – são os limites superiores da primeira, segunda e enésima dimensão da matriz, respectivamente.

- * **2^a etapa** – definição das variáveis que poderão conter este novo tipo de dado

<identificador 1>, ... <identificador n> : <nome do novo tipo>;

A figura 1.2 mostra a representação de uma estrutura matriz com duas dimensões – linha e coluna.

Figura 1.2 – Representação de estrutura Matriz 4x3

O diagrama mostra uma matriz 4x3 com todos os elementos igualados a zero. As linhas estão rotuladas com 1, 2, 3 e 4, e as colunas com 1, 2 e 3. Linhas e colunas são rotuladas com setas apontando para o lado direito e para baixo, respectivamente. A seta apontando para o interior da célula é rotulada "Conteúdo".

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

Fonte: Elaborada pelo autor.

Exemplo 1.6

Criar uma matriz 8x12 devidamente zerada.

```

inicio
    tipo M = matriz [1:8, 1:12] inteiro;
    MAT:M;
    I,J:inteiro;
    para I de 1 até 8 faca
        para J de 1 até 12 faca
            MAT[I,J] ← 0;
        fimpara;
    fimpara;
    para I de 1 até 8 faca
        para J de 1 até 12 faca
            imprima (MAT[I,J]);
        fimpara;
    fimpara;
fim.

```

Fonte: Kantek; De Bassi, (2013).

Exemplo 1.7

Dada uma matriz 7x13 devidamente preenchida, o algoritmo deve encontrar sua matriz transposta.

```
inicio
    tipo M1 = matriz [1:7,1:13] real;
    tipo M2 = matriz [1:13,1:7] real;
    MAT1:M1;
    MAT2:M2;
    I,J:inteiro
    para I de 1 até 7 faca
        para J de 1 até 13 faca
            leia (MAT1[I,J]);
            fimpara;
        fimpara;
    para I de 1 até 7 faca
        para J de 1 até 13 faca
            MAT2[J,I] ← MAT1[I,J];
            fimapara;
        fimpara;
    para I de 1 até 13 faca
        para J de 1 até 7 faca
            imprima(MAT2[I,J]);
            fimpara;
        fimpara;
    fim.
```

Fonte: Kantek; De Bassi (2013).

Exemplo 1.8

Ler dois vetores A e B, cada um com 7 elementos, e construir uma matriz C de mesma dimensão. A primeira coluna deverá ser formada pelos elementos de A e a segunda coluna pelos elementos de B.

```
Inicio
    Tipo V = vetor [1:7] real
    tipo M = matriz [1:7, 1:2] real;
    MAT:M;
    A, B:V;
    I,J:inteiro;
    para I de 1 até 7 faca
        leia (A[I]);
```

```

leia (B[I]);
MAT[I,1] ← A[I];
MAT[I,2] ← B[I];
fimpara;
para I de 1 até 2 faça
    para J de 1 até 7 faça
        imprima (MAT[I,J]);
    fimpara;
fimpara;
fim.

```

Fonte: Kantek; De Bassi, (2013).

Exemplo 1.9

Escrever um algoritmo que leia uma matriz quadrada de ordem 23 e totalize os elementos colocados abaixo da diagonal principal (inclusive esta), imprimindo o total ao final. Para resolver este algoritmo é importante saber que na diagonal principal de uma matriz quadrada os índices I e J são iguais.

```

Início
    tipo M = matriz [1:23, 1:23] real;
    MAT:M;
    I,J, SOMA:inteiro;
    SOMA ← 0;
    para I de 1 até 23 faça
        para J de 1 até 23 faça
            leia (MAT[I,J]);
            se I >= J //valores abaixo da diagonal principal inclusive esta
                então SOMA ← SOMA + MAT[I,J];
            fimse;
        fimpara;
        imprima (SOMA);
fim.

```

Fonte: Kantek; De Bassi (2013).

1.4 Estrutura de dados heterogêneos

Para a definição de estruturas complexas de dados, a maioria das linguagens de programação disponibilizam uma ferramenta capaz de definir uma

Estrutura de Dados

estrutura chamada registro, que é considerada como tipos de dados heterogêneos, por permitir armazenar vários dados de tipos diferentes em uma mesma estrutura. Trata-se de um aglomerado de dados que pode ser acessado no todo, ou a cada um de seus elementos individualmente. É uma estrutura de dados heterogênea. É um conjunto de dados logicamente relacionados, mas de tipos diferentes (numérico, literal, lógico), sob um único tipo abstrato de dado (TAD) (KANTEK; DE BASSI, 2013).

Registros correspondem a conjuntos de posições de memória conhecidos por um mesmo nome e individualizados por identificadores associados a cada conjunto de posições, os campos (KANTEK; DE BASSI, 2013).

Os tipos registros devem ser declarados antes das variáveis, pois poderá ocorrer a necessidade de se declarar uma variável com o tipo registro anteriormente atribuído. O procedimento é semelhante ao de vetores e matrizes (KANTEK; DE BASSI, 2013).

Sintaxe de declaração de registros:

- × o **primeiro passo** é a definição do tipo de dado/registro.

```
tipo <nome> = REGISTRO <identif 1> : tipo de dado;  
                      <identif 2> : tipo de dado;  
                      ....  
                      <identif n-1, identif n> : tipo de dado;  
fimregistro;
```

Os campos são identificadores de variáveis.

- ✗ o **segundo passo** é a definição das variáveis que poderão conter este novo tipo de dado.

<identificador 1>, ... <identificador n> : <nome do novo tipo>;

Figura 1.3 – Representação da estrutura registro

José Silva	Rua da Paz	230	fundos	99.999- 000	9999- 9999	8888- 8888	Registro
Nome	Rua	número	compl	Cep	fixo	celular	Campos

Endereço Telefone

Fonte: Elaborada pelo autor.

Para fazer acesso a um campo contido em um registro, precisa-se indicar todo o caminho necessário para encontrar a informação, separando-os por um ponto (.).

```
<nome do registro . identificador >
```

Desta forma, pode-se atribuir um valor a um campo do registro, imprimir tal valor e utilizar o valor em expressões.

Exemplo 1.10

Declarar um registro para armazenar um cadastro pessoal com os dados:

 nome, endereço, sexo, salário

```
Tipo REG1 = REGISTRO NOME : caractere [40];
                         END : caractere [80];
                         SEXO : inteiro;
                         SAL : real;

fimreg1;
CAD : REG1;

leia (CAD);
imprima (CAD.NOME);
imprima (CAD. END);
```

Também pode ter registro de registro. No exemplo anterior, inseriremos no campo END um registro contendo: rua, número, complemento e CEP, mostrado a seguir.

```
Tipo REG1 = registro NOME : caractere [40];
                         END : REG2;
                         SEXO : inteiro;
                         SAL : real;

fimreg1;
tipo REG2 = registro RUA : caractere [60];
                         NUM : inteiro;
                         COMPL : caractere [ 40];
                         CEP : inteiro;

fimreg2;
CAD : REG1;

leia (CAD);
Imprima (CAD. Nome);
```

Estrutura de Dados

```
imprima (CAD. END.CEP);  
imprima (CAD.SAL);
```

Quando necessário, é possível definir um campo de um registro como tendo um conteúdo vetor ou matriz. Outro conceito interessante é a definição de um vetor tendo como conteúdo uma estrutura registro. Esta estrutura é bastante útil para a definição de organização de dados.

No exemplo anterior, caso haja a necessidade de manusear o salário dos últimos 12 meses do funcionário, isto pode ser feito da seguinte maneira:

```
Tipo REG1 = registro NOME : caractere [40];  
                      END : REG2;  
                      SEXO : inteiro;  
                      SAL : V;  
  
fimreg1;  
tipo REG2 = registro RUA : caractere [60];  
                      NUM : inteiro;  
                      COMPL : caractere [ 40];  
                      CEP : inteiro;  
  
fimreg2;  
tipo V = vetor [1:12] real;  
CAD : REG1;  
I: inteiro;  
  
leia (CAD);           //acesso ao registro com todos os campos  
imprima (CAD. Nome);  
imprima (CAD. END.CEP);  
para I de 1 até 12 faça //imprime o salário dos 12 meses  
    imprima (CAD.SAL[I]);  
fimpara;
```

Síntese

Neste capítulo apresentamos as estruturas de dados mais simples, que são os vetores; as matrizes e os registros, construídos a partir dos tipos de dados básicos inteiro, real, caractere e lógico. Estas estruturas permitem o manuseio de diversas variáveis utilizando o mesmo nome de variável, e o acesso aos dados é feito por meio dos índices, no caso de vetores e matrizes, e de campos para os registros. As estruturas de dados facilitam a escrita dos algoritmos e melhoram a compreensão da lógica utilizada na resolução dos problemas.

Atividades

1. Escrever um algoritmo que leia um vetor contendo 10 valores. Calcule a média entre estes valores, e imprima: todos os valores do vetor, a média e a diferença entre cada valor individual e a média calculada.
2. Fazer um algoritmo para corrigir provas de múltipla escolha. Cada prova tem 10 questões, cada questão valendo um ponto. O primeiro conjunto de dados a ser lido será um vetor do gabarito para a correção da prova. Os outros dados serão os números dos alunos e suas respectivas respostas e o último número, de um aluno fictício, será 9999.

O algoritmo deve calcular e imprimir:

- a) para cada aluno, seu número e sua nota;
 - b) a porcentagem de aprovação, sabendo-se que a nota mínima de aprovação é 6;
 - c) a nota que teve maior frequência absoluta, ou seja, a nota que apareceu maior número de vezes (supondo a inexistência de empates).
3. Dado um tabuleiro de xadrez TAB no qual, para facilitar a indicação das pedras, vamos convencionar:

- ✗ 1 – peão
- ✗ 2 – cavalos
- ✗ 3 – torres
- ✗ 4 – bispos
- ✗ 5 – reis
- ✗ 6 – rainhas
- ✗ 0 – ausência de pedra

Ler os dados e contar a quantidade de cada tipo de peça no tabuleiro em um certo momento do jogo. Tamanho do tabuleiro: 8x8.

4. Uma seguradora possui um cadastro de seus segurados contendo registros com os campos:

Estrutura de Dados

- ✗ Matrícula – inteiro;
- ✗ Nome – caractere[40];
- ✗ Data de nascimento – dia, mês e ano;
- ✗ Sexo – caractere [1].

Fazer um algoritmo que leia tais registros até que seja lida uma matrícula = 0. Este algoritmo deve calcular a idade do segurado, com base na data atual, em dias. Ao final, devem ser gerados os seguintes registros:

HOMENS Nome: caractere [40]

Idade: inteiro

MULHERES Nome: caractere [40];

Idade: inteiro;

2

Estruturas de Dados Elementares: Pilha

NO MUNDO REAL existem situações que, para serem tratadas por meio de um algoritmo ou de um programa, precisam, além de uma estrutura de dados para a sua representação e um método de resolução, alguma característica adicional que é específica da situação em questão. Por exemplo: os dados estarem ordenados em ordem crescente, existir uma prioridade na recuperação da informação, entre outras situações.

MUITAS VEZES ESTAS situações não possuem um único tipo de estrutura de dados, podendo ser representadas por diferentes formas. Entretanto, o método, ou seja, a maneira como elas devem ser tratadas não se modifica, independentemente de como são representadas.

Utilizamos certas estruturas de dados elementares chamadas de **PILHA**, **FILA** e **LISTA** para representar e trabalhar estas situações particulares.

Podemos construir essas estruturas de dados elementares a partir de outras estruturas de dados como vetores ou estruturas encadeadas. Estas estruturas devem conter, além das informações de interesse, uma referência ao elemento anterior e/ou próximo da estrutura, implementada por meio de registros ou de objetos (Programação Orientada a Objetos – POO).

Objetivos de aprendizagem:

1. conhecer estruturas de dados Pilha;
2. conhecer a representação e funcionamento das estruturas de dados Pilha;
3. aplicar esta nova estrutura de dados na solução de problemas.

2.1 Estrutura Pilha

Uma estrutura de dados tipo Pilha representa uma situação na qual temos uma ordem de acesso e retirada de elemento a partir de uma única extremidade da estrutura de dados, chamada de **TOPO**. Segundo Tenenbaum, Langsan e Augenstein (1995, p. 86), Pilha é “um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados itens em uma extremidade chamado topo da Pilha”.

Empilhar papéis, empilhar caixas em um depósito ou empilhar pratos são atividades comuns em nosso cotidiano. Consideremos um exemplo comum: uma pilha de pratos lavados representando um exemplo de Pilha. Normalmente, quando lavamos um prato, empilhamos o prato no topo da pilha de pratos lavados. Se não desejarmos correr o risco de quebrar algum prato, devemos colocar um novo prato na parte de cima da pilha. Dessa forma, somente podemos inserir um novo elemento em uma das extremidades da Pilha, o seu **TOPO**.

Da mesma forma, se não desejarmos correr novamente o risco de quebrar algum prato da pilha, normalmente quando retiramos um prato lavado

para enxugar e guardar devemos retirar o prato que estiver na parte de cima da Pilha, ou seja, devemos retirar o último prato colocado. Dessa forma, somente podemos excluir um elemento da Pilha da sua extremidade designada por **TOPO**.

Na estrutura tipo Pilha, somente temos acesso ao último elemento. Uma vez que este elemento tenha sido retirado, perdemos por completo a informação deste elemento. A informação dos demais elementos que estiverem na Pilha está guardada. Podemos até verificar quais são os elementos que estão na Pilha, mas o único elemento com permissão de acesso é o elemento que estiver no **TOPO** da fila (o último elemento).

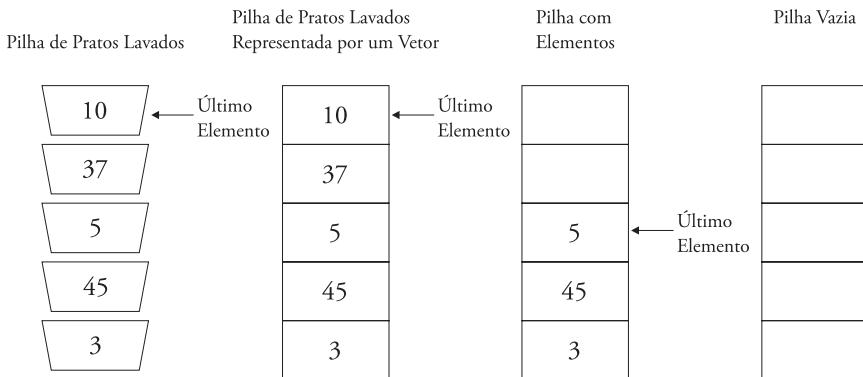
Essas estruturas de dados com este tipo de organização são conhecidas pela sigla em inglês **LIFO** (*Last In, First Out*), que significa “o último a entrar é o primeiro a sair”.

Para representar uma estrutura tipo Pilha por meio de um vetor, utilizamos um índice para determinar o acesso ao vetor para inserir e retirar elementos da Pilha. Uma propriedade fundamental de um vetor é que podemos acessar qualquer uma de suas posições, bastando para isto fornecer o número da posição desejada. No entanto, não é isto que queremos construir com uma Pilha. Para utilizar um vetor para implementar uma Pilha devemos restringir o acesso ao vetor, impondo uma regra para inserir ou retirar elementos do vetor. Portanto, devemos elaborar um algoritmo que faça esta restrição ao uso do vetor no momento de inserir e excluir elementos. Estas operações são denominadas de empilhamento e desempilhamento, respectivamente.

Para implementar um exemplo de um algoritmo para uma Pilha, vamos criar uma estrutura de dados declarando um vetor de MAX posições, no qual desejamos inserir como informação, que será apenas um número do tipo inteiro. Esta situação pode representar a nossa Pilha de pratos lavados, sendo que os pratos foram marcados como números. Representamos algumas situações possíveis para uma Pilha na figura 1.1. Nesta representação, a indicação do último elemento da Pilha representa o TOPO da Pilha. Por meio da indicação da posição do topo da Pilha é possível empilhar e desempilhar elementos (pratos lavados) da Pilha.

Estrutura de Dados

Figura 2.1 – Representação de uma estrutura de dados elementar tipo Pilha



Fonte: Elaborada pelo autor.

Em uma estrutura Pilha, a variável **TOPO** é um índice, e indica a posição no vetor na qual um elemento deve ser inserido. Se **TOPO** for maior do que **MAX**, esta situação fornece a condição de Pilha **CHEIA**, ou seja, não é possível inserir novos elementos na Pilha. A condição de Pilha **VAZIA** é dada quando **TOPO** for igual a um (primeira posição), não sendo possível excluir nenhum elemento da Pilha. Portanto, o controle de acesso ao vetor é feito por meio da variável **TOPO**. Desta forma, o índice **TOPO** está apontando sempre para a posição de inserção.

2.2 Algoritmos para manuseio da estrutura Pilha

A estrutura Pilha em sua representação é basicamente um vetor, porém é o padrão de comportamento desta estrutura que irá definir seu uso. No caso da estrutura Pilha, o padrão de comportamento é definido pelo LIFO: o último a entrar é o primeiro a sair. O padrão de comportamento desta estrutura será dado pelo algoritmo de empilhamento e desempilhamento.

Quando criamos os algoritmos que determinam o padrão de comportamento da Pilha, é importante verificar antes situações de Pilha cheia e Pilha

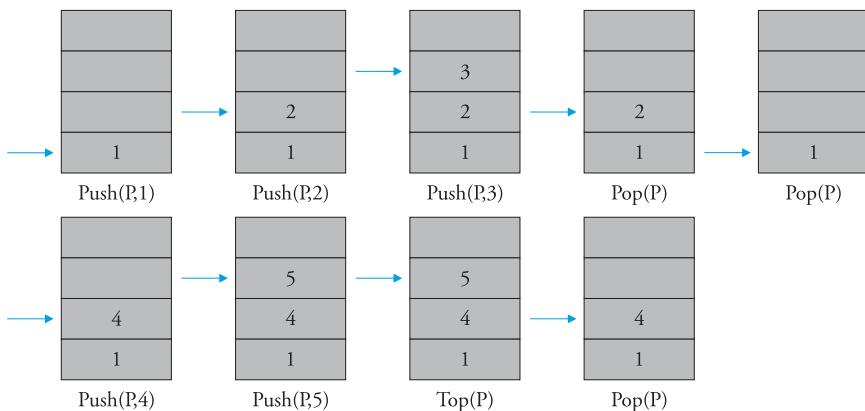
vazia. Por exemplo: em uma Pilha cheia não é possível empilhar mais valores (não cabe mais prato na Pilha); em uma Pilha vazia não existem valores para serem desempilhados (não existem pratos na Pilha para serem retirados).

Uma Pilha normalmente suporta 3 operações básicas:

- ✖ **push** – empilhar onde se insere um novo elemento na Pilha, que ocorre sempre no topo da Pilha;
- ✖ **pop** – desempilhar, que remove um elemento da Pilha. Também ocorre sempre no topo da Pilha;
- ✖ **top** – retorna o elemento contido no topo da Pilha.

Na figura 1.2, que representa um exemplo de uso de Pilha, foram realizadas as operações apresentadas na tabela 1.1. Na figura, a seta indica o topo da Pilha, os números são os conteúdos da Pilha e abaixo de cada Pilha a operação que está sendo realizada. Foram realizadas 9 operações na Pilha que inicia vazia. A tabela 1.1 mostra as operações que foram realizadas, como se apresenta a Pilha e os valores que são retornados após cada operação. Pode ser observado que algumas operações não apresentam retorno, apesar de alterar a apresentação da Pilha.

Figura 2.2 – Resultado da aplicação das operações na Pilha P



Fonte: Elaborada pelo autor.

Estrutura de Dados

Tabela 1.1 – Operações realizadas na Pilha P

Operação	Formação da Pilha	Retorno da operação
-----	P:[]	Sem retorno
Push [P,1]	P:[1]	Sem retorno
Push [P,2]	P:[2,1]	Sem retorno
Push [P,3]	P:[3,2,1]	Sem retorno
Pop [P]	P:[2,1]	3
Pop [P]	P:[1]	2
Push [P,4]	P:[4,1]	Sem retorno
Push [P,5]	P:[5,4,1]	Sem retorno
Top [P]	P:[5,4,1]	5
Pop [P]	P:[4,1]	5

Fonte: Elaborada pelo autor.

Na sequência, são apresentados os algoritmos para manuseio da Pilha. Nestes algoritmos está sendo considerada uma estrutura vetor V com no máximo 5 elementos inteiros. Segue a definição da estrutura de dados Pilha:

```
CONST MAX = 5
CONST TRUE = 1
TIPO VET = VETOR[MAX] : inteiro;
TOPO : inteiro;
```

2.2.1 Algoritmo para empilhar valor na Pilha

Para empilhar um novo valor na Pilha, é preciso verificar se a Pilha não está cheia (TOPO < MAX) para então inserir o novo valor e incrementar o valor de TOPO, indicando a próxima posição a ser ocupada na Pilha.

```
procedimento empilhar(V : VET)
início
    VALOR : inteiro;
    se TOPO > MAX
        então
```

```

        imprima("PILHA CHEIA!");
        empilhar;
fimse
imprima("ENTRE COM UM NÚMERO : ");
leia(VALOR);
V[TOPO] ← VALOR;
TOPO ← TOPO + 1;

Fim // procedimento empilhar

```

2.2.2 Algoritmo para desempilhar valor da Pilha

Para desempilhar um valor da Pilha, é preciso verificar se a Pilha não está vazia ($\text{TOPO} = 1$), pois uma Pilha vazia não terá valor para ser desempilhado. Então pode-se retirar um valor da Pilha e decrementar o valor de TOPO , indicando que a posição está vaga e pode ser ocupada.

```

procedimento desempilhar(V : VET)
início
    se  $\text{TOPO} = 1$ 
        então
            imprima("PILHA VAZIA!");
            desempilhar;
        fimse;
     $\text{TOPO} \leftarrow \text{TOPO} - 1$ ;
Fim; // procedimento desempilhar

```

2.2.3 Algoritmo para imprimir valores da Pilha

Outra atividade comum de ser realizada com a estrutura Pilha é a de imprimir seus elementos. A impressão percorre a Pilha do Topo até a primeira posição do vetor que representa a Pilha, mostrando seus valores. Lembrando que uma Pilha vazia não tem valores a serem impressos ($\text{TOPO} = 1$).

```

procedimento imprimir_pilha(V : VET)
início
    J : inteiro;
    imprima("PILHA : ");
    se  $\text{TOPO} = 1$ 
        então

```

Estrutura de Dados

```
    imprima("PILHA VAZIA!");
senão
    para J de TOPO - 1 até 1 passo - 1 faça
        imprima(V[J]);
        se J = TOPO - 1
            então
                imprima("← TOPO");
            fimse;
            imprima();
        fimpara;
    fimse;
fim; // procedimento imprimir Pilha
```

Na sequência, é apresentado o algoritmo completo com os procedimentos que manuseiam a estrutura Pilha.

```
algoritmo //PILHA para inteiros utilizando um vetor
CONST MAX = 5
CONST TRUE = 1
TIPO VET = VETOR[MAX] : inteiros;
TOPO : inteiro;

procedimento empilhar(V : VET);
procedimento desempilhar (V : VET);
imprimir_pilha(V : VET);

inicio
    PILHA : VET;
    OPÇÃO : inteiro;
    TOPO ← 1;
    repita
        imprima("1. INSERIR");
        imprima("2. EXCLUIR");
        imprima("3. SAIR");
        imprimir_pilha(PILHA);
        imprima("ENTRE COM SUA OPÇÃO");
        leia(OPÇÃO);
        escolha(OPÇÃO)
        caso 1 : empilhar(PILHA);
        fimcaso;
        caso 2 : desempilhar(PILHA);
```

```

        fimcaso;
    caso 3 : fim; //algoritmo
    fimescolha;
    enquanto(TRUE);
fim. //algoritmo

procedimento empilhar(V : VET)
início
    VALOR : inteiro;
    se TOPO > MAX
        então
            imprima("PILHA CHEIA!");
            empilhar;
        fimse;
        imprima("ENTRE COM UM NÚMERO : ");
        leia(VALOR);
        V[TOPO] ← VALOR;
        TOPO ← TOPO + 1;
    fim; // procedimento empilhar

procedimento desempilhar(V : VET)
início
    se TOPO = 1
        então
            imprima("PILHA VAZIA!");
            desempilhar;
        fimse;
        TOPO ← TOPO - 1;
    fim; // procedimento desempilhar

procedimento imprimir_pilha(V : VET)
início
    J : inteiro;
    imprima("PILHA : ");
    se TOPO = 1
        então
            imprima("PILHA VAZIA!");
        senão
            para J de TOPO - 1 até 1 passo - 1 faça

```

```
    imprima(V[J]);
    se J = TOPO - 1
        então
            imprima("← TOPO");
        fimse;
        imprima( );
    fimpara;
fimse;
fim; // procedimento imprimir_pilha
```

2.3 Utilização prática da estrutura Pilha

O conceito de Pilha é amplamente utilizado na computação, principalmente no controle de execução de processo pelo sistema operacional. Um uso bastante comum é o controle de chamada de função durante a execução de um programa pelo processador. Cada vez que uma função chama outra função, a função que foi chamada é empilhada para ser executada. Ao término da execução desta segunda função, ela é desempilhada, para que primeira função chamadora continue sua execução. Se, durante a execução desta segunda função, houver uma chamada para uma terceira função, há um novo empilhamento. Quando acontece de em um programa possuir mais empilhamentos de funções do que a memória do computador comporta, diz-se que houve um “estouro de Pilha”, ou “*stack overflow*”.

Outra forma bastante básica para o uso de Pilha na computação é no controle dos comandos SE – então – senão – FIMSE aninhados. O compilador normalmente utiliza uma estrutura de Pilha para controlar o número de comandos SEs e seus respectivos FIMSEs que foram abertos e fechados. Cada SE aberto é empilhado e cada FIMSE encontrado desempilha o respectivo SE. Ao final da compilação a Pilha de controle de SEs aninhados deve estar zerada (TOPO = 1). Caso não esteja, existe algum SE sem seu respectivo FIMSE ou ao contrário. Da mesma forma é possível controlar outros comandos aninhados.

Esta forma de controle também é utilizada pelo compilador para verificar os parênteses (“”) utilizados em expressões de atribuição ou condições dos comandos. Cada parêntese aberto é empilhado e cada parêntese fechado desempilha seu respectivo parêntese aberto. Se ao final da verificação a Pilha

não estiver vazia existe algum parêntese que não foi fechado ou um parêntese fechado que está sobrando. Desta forma é possível validar duas situações:

1. existe um número igual de parênteses abertos e fechados;
2. todo parêntese fechado está precedido por um parêntese aberto correspondente.

É possível então verificar que expressões como:

- ✗ $((X + Y) \quad \text{ou} \quad X + Y)$ – não satisfazem o item 1
- ✗ $)X + Y (- Z \quad \text{ou} \quad (X + Y)) - (Z + W)$ – satisfazem o item 1 mas violam o item 2.

Um exemplo clássico do uso de Pilha é na solução do problema da Torre de Hanói (TENENBAUM; LANGSAN; AUGENSTEIN, 1995). A Torre de Hanói é uma espécie de quebra-cabeça no qual existem três estacas e cinco discos de tamanhos diferentes (figura 1.3). Os discos são colocados uns sobre os outros, sendo que nunca um menor fica abaixo de um maior. O problema consiste em passar todos os discos de uma estaca para qualquer uma das outras estacas, usando uma das estacas como auxiliar. Neste problema, cada um dos pinos pode ser considerado uma Pilha. Aproveite para pensar na solução deste problema clássico!

Figura 2.3 – Torre de Hanói



Fonte: Shutterstock.com/Dmitry Elagin.

Síntese

Conforme verificado, as estruturas de dados elementares aplicam um padrão de comportamento sobre as estruturas de dados. Este padrão de comportamento é utilizado nos algoritmos que operam estas estruturas de dados. Neste capítulo foi comentado sobre a estrutura de dados elementar Pilha. O padrão de comportamento da Pilha é semelhante ao que utilizamos no mundo real, ou seja, tanto a inserção quanto a remoção ocorrem no topo da Pilha (*LIFO – last in/first out*). Para realizarmos as operações na Pilha é importante conhecer a posição do topo da Pilha. As operações básicas da Pilha são push (empilhar), pop (desempilhar) e top (topo). Neste capítulo, utilizamos a estrutura de dados vetor para representar a Pilha.

Atividades

1. Escrever um algoritmo que leia uma Pilha contendo 1000 valores e exclua os valores negativos da Pilha, gerando uma segunda Pilha e mantendo a ordem dos valores da Pilha original.
2. Escrever um algoritmo que inverta a ordem dos elementos de uma Pilha. O algoritmo deve ler uma pilha contendo 100 valores e, utilizando apenas uma estrutura auxiliar, inverter os elementos da Pilha lida.
3. Escrever um algoritmo que leia uma Pilha contendo 500 valores inteiros e construa duas Pilhas, onde a primeira Pilha contenha os valores pares e a segunda Pilha contenha os valores ímpares da Pilha original. É importante manter a ordem original dos elementos nas Pilhas pares e ímpares.
4. Escrever um algoritmo que leia uma Pilha contendo 250 nomes de cidades brasileiras e troque de lugar os valores do topo e da base da Pilha, mantendo a ordem original dos demais valores da Pilha.

3

Estrutura de Dados Elementares: Fila

UMA FILA é um conceito com o qual nos deparamos com frequência em nosso cotidiano. Fazemos Fila para entrar no ônibus, sermos atendidos no caixa do banco, para nos servir no restaurante por quilo... enfim, a Fila serve para organizar o atendimento. A forma de organização de uma Fila é conhecida: o primeiro que chega é o primeiro a ser atendido. Iremos utilizá-la neste capítulo.

Objetivos de aprendizagem:

1. conhecer e compreender as estruturas de dados Fila;
2. conhecer a representação e funcionamento das estruturas de dados Fila;
3. aplicar esta nova estrutura de dados na solução de problemas.

3.1 Estrutura Fila

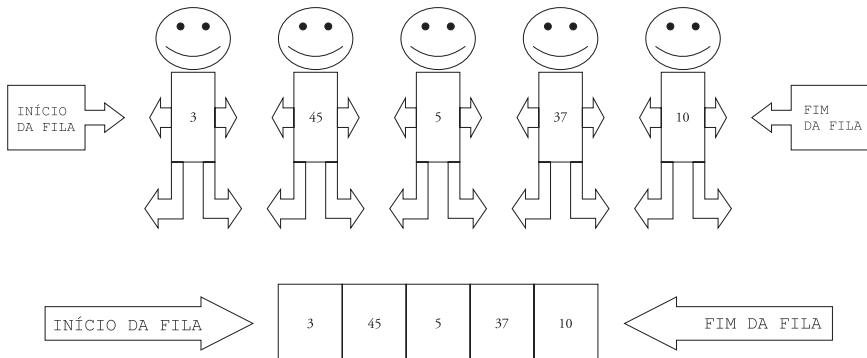
Segundo Forbellone e Eberspacher (2005), as Filas, na abordagem computacional, são “estruturas de dados que se comportam como as Filas que conhecemos [...] cuja finalidade principal é registrar a ordem de chegada de seus componentes.”. Neste contexto, a fila do caixa no supermercado é um exemplo de estrutura de dados tipo Fila. Quando uma pessoa se dirige ao caixa, entra no final da fila e aguarda para ser atendida. A pessoa que estiver no início da fila será a primeira pessoa a ser atendida. Portanto, na Fila um novo elemento será inserido na extremidade da estrutura denominada *fim* da Fila. Um elemento a ser excluído deve ser aquele que estiver na extremidade da estrutura denominada *início* da Fila. Podemos verificar quais são os elementos que estão na Fila, porém os elementos que estão nas extremidades *início* e *fim* são os elementos que estão nas posições de acesso na Fila para excluir (o primeiro elemento) e inserir (a partir deste último elemento), respectivamente.

A estrutura assim estabelecida é conhecida como uma Fila linear, também designada pela sigla em inglês FIFO (*first in, first out*) que significa “o primeiro a entrar será o primeiro a sair”, ou LIFO (*Last In, Last Out*) que significa “o último a entrar será o último a sair”. Estas siglas indicam basicamente a mesma ordem de utilização da estrutura. Entretanto, a Fila linear implementada por meio de um vetor possui uma restrição: quando utilizamos todas as posições do vetor, não podemos inserir novos elementos.

Para implementar um exemplo de algoritmo para uma Fila, vamos criar uma estrutura de dados declarando um vetor de *max* posições, na qual desejamos inserir como informação apenas um número do tipo inteiro. Esta situação pode representar a nossa fila do caixa do supermercado, na qual cada cliente possui um número de identificação. Os índices *início* e *fim* fornecem a posição no vetor que deve ser utilizada para excluir e incluir elementos,

respectivamente, e são inicializados com o valor da primeira posição do vetor. Representamos uma Fila na figura 3.1 indicando início e o fim da Fila.

Figura 3.1 - Representação de uma estrutura de dados elementar tipo Fila Linear



Fonte: Elaborada pelo autor.

3.2 Algoritmos para manuseio da estrutura Fila linear

A estrutura Fila linear em sua representação é basicamente um vetor, porém é o padrão de comportamento desta estrutura que irá definir seu uso. No caso da estrutura Fila linear o padrão de comportamento é definido pelo LIFO (*last in, last out*): o último a entrar é o último a sair. O padrão de comportamento desta estrutura será dado pelo algoritmo de enfileirar e desenfileirar.

Quando criamos os algoritmos que determinam o padrão de comportamento da Fila é importante verificar antes situações de Fila cheia e Fila vazia. Por exemplo, em uma Fila cheia não é possível enfileirar mais valores; em uma Fila vazia não existem valores para serem desenfileirados.

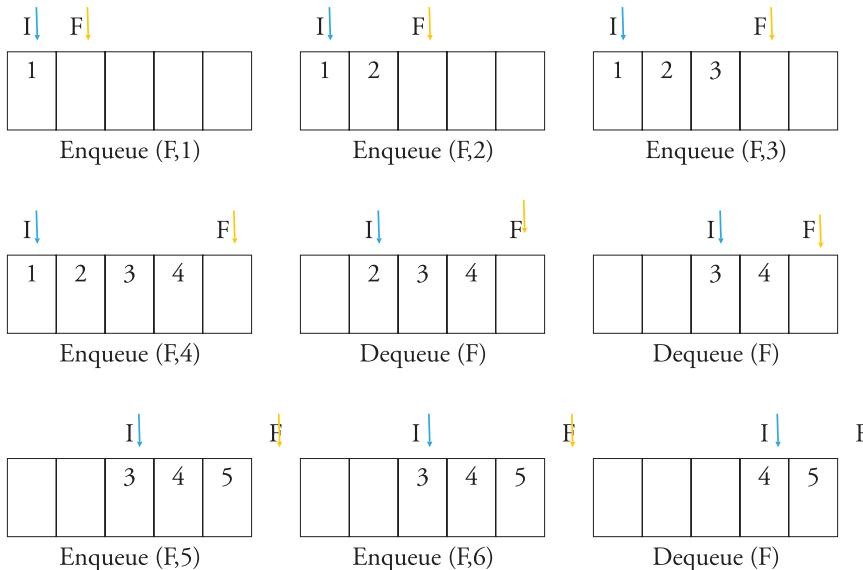
A Fila também possui operações básicas que são denominadas:

- ✖ enqueue – ou enfileirar, onde um novo elemento é inserido sempre no final da Fila;
- ✖ dequeue – ou desenfileirar, onde o elemento é removido da Fila e sempre ocorre no começo da Fila.

Estrutura de Dados

Da mesma forma que foi feito com a Pilha, serão mostrados na figura 3.2 exemplos de uso de Fila a partir das operações apresentadas na tabela 3.1.

Figura 3.2 - Resultado da aplicação das operações na Fila F



Fonte: Elaborada pelo autor.

Tabela 3.1 - Operações realizadas na Fila F

Operação	Formação da Fila	Retorno da operação
-----	F:[]	Sem retorno
Enqueue [F,1]	F:[1]	Sem retorno
Enqueue [F,2]	F:[1,2]	Sem retorno
Enqueue [F,3]	F:[1,2,3]	Sem retorno
Enqueue [F,4]	F:[1,2,3,4]	Sem retorno
Dequeue [F]	F:[2,3,4]	1
Dequeue [F]	F:[3,4]	2
Enqueue [F,5]	F:[3,4,5]	Sem retorno

Operação	Formação da Fila	Retorno da operação
Enqueue [F,6]	F:[3,4,5]	Fila cheia
Dequeue [F]	F:[4,5]	3

Fonte: Elaborada pelo autor.

Na sequência são apresentados os algoritmos de manuseio da Fila. Da mesma forma que na Pilha, nestes algoritmos está sendo considerada uma estrutura vetor V com no máximo 5 elementos inteiros. A definição da estrutura de dados Fila segue:

```
CONST MAX = 5
CONST TRUE = 1
TIPO VET = VETOR[MAX] : inteiro;
INÍCIO, FIM : inteiro;
```

3.2.1 Algoritmo para enfileirar valor na Fila linear

Para enfileirar um novo valor na Fila é preciso verificar se a Fila não está cheia ($fim > max$), para então inserir um novo valor e incrementar o valor de fim , indicando a próxima posição a ser ocupada na Fila.

```
procedimento enfileirar (V : VET)
    inicio
        VALOR : inteiro;
        se FIM > MAX
            então
                imprima("FILA CHEIA!");
                enfileirar;
            fimse;
        imprima("ENTRE COM UM NÚMERO : ");
        leia(VALOR);
        V[FIM] ← VALOR;
        FIM ← FIM + 1;
    Fim; // procedimento enfileirar
```

3.2.2 Algoritmo para desenfileirar valor da Fila linear

Para desenfileirar um valor da Fila é preciso verificar se a Fila não está vazia ($início = 0$ ou $início = fim$), pois uma Fila vazia não terá valor para ser

Estrutura de Dados

desenfileirado. Depois disso, pode-se então retirar o valor da Fila e incrementar o valor de *início*, indicando que a posição está vaga e pode ser ocupada.

```
procedimento desenfileirar(V : VET)
    inicio
        se INÍCIO = FIM
            então
                imprima("FILA VAZIA!");
                desenfileirar;
            fimse;
        INÍCIO ← INÍCIO + 1;
    Fim; //procedimento desenfileirar
```

3.2.3 Algoritmo para imprimir valores da Fila linear

Outra atividade comum a ser realizada com a estrutura Fila é o de imprimir seus elementos. A impressão percorre a Fila do *início* até o *fim* do vetor que representa a Fila mostrando seus valores. Não esquecendo que uma Fila vazia não tem valores a serem impressos (*início* = 0 ou *início* = *fim*).

```
procedimento imprimir_fila(V : VET)
    inicio
        J : inteiro;
        imprima("INÍCIO DA Fila : ");
        se INÍCIO = FIM
            então
                imprima("FILA VAZIA!");
            senão
                para J de INÍCIO até FIM - 1 faça
                    imprima(V[J]);
                fimpara;
        fimse;
        imprima("FIM DA Fila");
    Fim; // procedimento imprimir_fila
```

Na sequência é apresentado o algoritmo completo com os procedimentos que manuseiam a estrutura Fila.

```
algoritmo //FILA para inteiros utilizando um vetor
    CONST MAX = 5
    CONST TRUE = 1
    TIPO VET = VETOR[MAX] : inteiro;
    INÍCIO, FIM : inteiro;
```

```

procedimento enfileirar(V : VET);
procedimento desenfileirar(V : VET);
imprimir_fila(V : VET):
início
    FILA : VET;
    OPÇÃO : inteiro;
    (INÍCIO, FIM) ← 1;
    Repita
        imprima("1. INSERIR");
        imprima("2. EXCLUIR");
        imprima("3. SAIR");
        imprimir_fila(FILA);
        imprima("ENTRE COM SUA OPÇÃO");
        leia(OPÇÃO);
        escolha(OPÇÃO)
            caso 1 : enfileirar(FILA);
            fimcaso;
            caso 2 : desenfileirar(FILA);
            fimcaso;
            caso 3 : fim; //algoritmo
        fimescolha;
        enquanto(TRUE);
    fim. //algoritmo

procedimento enfileirar(V : VET)
início
    VALOR : inteiro;
    se FIM > MAX
        então
            imprima("FILA CHEIA!");
            enfileirar;
        fimse.
    imprima("ENTRE COM UM NÚMERO : ");
    leia(VALOR);
    V[FIM] ← VALOR;
    FIM ← FIM + 1;
Fim; // procedimento enfileirar

procedimento desenfileirar(V : VET)
início
    se INÍCIO = FIM
        então
            imprima("FILA VAZIA!");
            desenfileirar;

```

Estrutura de Dados

```
fimse
    INÍCIO ← INÍCIO + 1;
Fim; //procedimento desenfileirar

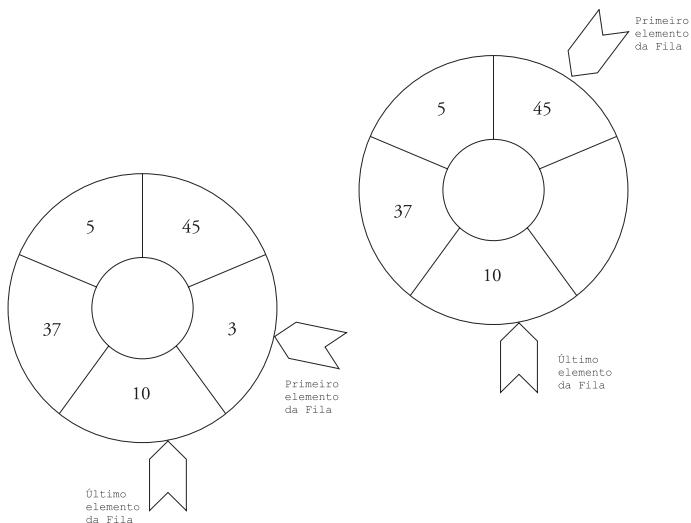
procedimento imprimir_fila(V : VET)
início
    J : inteiro;
    imprima("INÍCIO DA Fila : ");
    se INÍCIO = FIM
        então
            imprima("FILA VAZIA!");
        senão
            para J de INÍCIO até FIM - 1 faça
                imprima(V[J]);
            fimpara;
    fimse;
    imprima("FIM DA Fila");
fim; // procedimento imprimir_fila
```

Um dos problemas que podemos verificar na estrutura de dados Fila linear é quando o número de elementos da Fila atinge o número de elementos máximo do vetor, a Fila torna-se cheia e não é possível reutilizar a estrutura.

3.3 Estrutura de dados Fila circular

Podemos aperfeiçoar o algoritmo da Fila linear permitindo que o algoritmo utilize as posições do vetor que estiverem disponíveis quando chegarmos no final da Fila. Para isto iremos controlar o número de elementos inseridos na Fila e permitir que os índices *início* e *fim* (que controlam a posição de enfileirar e desenfileirar, respectivamente) retornem à primeira posição quando chegarem ao limite máximo do vetor. Ilustramos uma Fila linear transformada em uma Fila circular na figura 3.3, indicando o primeiro e o último elementos da Fila. Na verdade, a Fila circular continua sendo uma Fila linear com a possibilidade de utilizar as posições que foram liberadas. A representação feita na figura 3.3 é apenas um recurso didático para se imaginar uma Fila linear como sendo circular.

Figura 3.3 - Representação de uma estrutura de dados elementar tipo Fila circular



Fonte: Elaborada pelo autor.

Com a solução de Fila circular, as variáveis de controle agora são *início*, *fim* e *elementos*. A função da variável *elementos* é indicar quantas posições da Fila estão efetivamente ocupadas. De qualquer forma, é preciso alterar o valor das variáveis *início* e *fim*. Em função disto, no procedimento *enfileirar* é preciso inserir novos comandos, estas alterações estão mostradas em destaque no algoritmo a seguir:

1. comando para incrementar o valor de *elementos* da Fila;
2. comando para verificar se o número máximo de posições do vetor que representa a Fila foi excedido. Em caso verdadeiro (*fim* > *max*), o valor de *fim* irá ocupar o início da Fila (*fim* \leftarrow 1) o que irá caracterizar a Fila como circular conforme representado na figura 3.3.

Da mesma forma, no procedimento *desenfileirar* os comandos inseridos, também mostrados em destaque no algoritmo a seguir, são:

1. comando para decrementar o valor de *elementos* da Fila;

Estrutura de Dados

2. comando para verificar se o valor do *índice* excedeu o número máximo de posições da Fila. Em caso verdadeiro (*índice* > *max*), o valor de *índice* volta a ocupar a primeira posição da Fila (*índice* ← 1) o que irá caracterizar a Fila como circular.

Na sequência é apresentado o algoritmo de Fila circular com as alterações necessárias em destaque.

algoritmo // Fila CIRCULAR para inteiros utilizando um vetor

```
CONST MAX = 5
CONST TRUE = 1
TIPO VET = VETOR[MAX] : inteiros;
INÍCIO, FIM, ELEMENTOS : inteiro;
procedimento enfileirar(V : VET);
procedimento desenfileirar(V : VET);
imprimir_fila(V : VET);

inicio
    FILA : VET;
    OPÇÃO : inteiro;
    INÍCIO, FIM ← 1;
    ELEMENTOS ← 0;
repita
    imprima("1. INSERIR");
    imprima("2. EXCLUIR");
    imprima("3. SAIR");
    imprimir_fila(FILA);
    imprima("ENTRE COM SUA OPÇÃO");
    leia(OPÇÃO);
    escolha(OPÇÃO)
        caso 1 : enfileirar(FILA);
        fim;{caso}
        caso 2 : desenfileirar(FILA);
        fim;{caso}
        caso 3 : fim{algoritmo}
    fim;{escolha}
enquanto(TRUE);
fim. // algoritmo
```

```
procedimento enfileirar(V : VET)
inicio
    VALOR : inteiro;
    se ELEMENTOS = MAX
        então
```

```

        imprima("FILA CHEIA!");
        enfileirar;
fim; {se}
imprima("ENTRE COM UM NÚMERO : ");
leia(VALOR);
V[FIM] ← VALOR;
FIM ← FIM + 1;
se FIM > MAX
então
    FIM ← 1;
fim // Fila cheia e FIM ocupará a posição 1 do vetor formando o circulo
ELEMENTOS ← ELEMENTOS + 1; // aumenta o número de elementos da Fila
Fim; // procedimento enfileirar

procedimento desenfileirar(V : VET)
início
    se ELEMENTOS = 0
        então
            imprima("FILA VAZIA");
            desenfileirar;
fim; {se}
INÍCIO ← INÍCIO + 1;
se INÍCIO > MAX
então
    INÍCIO ← 1;
fim; // início chegou ao final do vetor então INICIO volta a ocupar a posição 1,
// formando o circulo
ELEMENTOS ← ELEMENTOS - 1; // diminui o número de elementos da Fila
Fim; // procedimento desenfileirar

procedimento imprimir_fila(V : VET)
início
    CONT, J : inteiro;
    imprima("INÍCIO DA Fila : ");
    se ELEMENTOS = 0
        então
            imprima("FILA VAZIA!");
    senão
        J ← INÍCIO;
        CONT ← 0;
        repita
            imprima(V[J]);
            J ← J + 1;
            CONT ← CONT + 1;

```

```
    se J > MAX
        então
            J ← 1;
        fim; {se}
    enquanto (CONT < ELEMENTOS)
        imprima("FIM DA Fila : ");
    fim; //procedimento imprimir_fila
```

3.4 Utilização da estrutura Fila na prática

Um exemplo comum de uso de Fila na computação acontece nas impressoras, quando solicitamos a impressão de mais de um documento ao mesmo tempo no computador conectado à impressora. E também nas impressoras compartilhadas com mais de um usuário, em rede por exemplo. Em ambos os casos, quando enviamos um documento para ser impresso, este entra no final da Fila de impressão e o documento impresso será o que estiver na posição do início da Fila.

O sistema operacional também utiliza o conceito de Fila para gerenciar o atendimento dos processos que precisam ser executados, quando não são priorizados – o primeiro processo que foi iniciado é o primeiro que será atendido e, desta forma, será o primeiro a ser finalizado.

Podemos perceber então que, em qualquer aplicação computacional que necessite simular uma Fila, isto é, cuja finalidade é registrar a ordem de chegada, a estrutura de dados Fila é aplicada.

Síntese

Neste capítulo foi discutido o tema estrutura de dados em Fila. A Fila segue o padrão de comportamento que conhecemos no cotidiano para organizar a ordem de chegada, logo o primeiro a chegar é o primeiro a sair (FIFO – *first in/first out*). A estrutura Fila linear aborda duas operações básicas – *enqueue* (enfileirar) e *dequeue* (desenfileirar). Para otimizar o comportamento da Fila linear, a Fila circular utiliza ainda a informação de número de elementos da Fila e permite que os valores de início e final da Fila sejam remanejados para reutilizar posições iniciais já desocupadas, agregando o conceito abstrato

de circular. Ambas as Filas, linear e circular, neste capítulo, utilizam a estrutura de dados vetor.

Atividades

1. Escreva um algoritmo que leia uma Fila com 200 nomes de candidatos a uma vaga de estágio em TI e inverta a ordem destes nomes na Fila.
2. Escreva um algoritmo que leia uma Fila com 1000 números inteiros e retire da Fila os números múltiplos de 7, mantendo a ordem da Fila original.
3. Escreva um algoritmo que leia uma Fila de 2000 posições contendo cada posição o número de matrícula e data de nascimento de adolescentes do Clube de Voleibol; caso o adolescente tenha atingido a maioridade (18 anos) ele deve ser retirado da Fila, mantendo a ordem da Fila original.

4

Ponteiros e Alocação Dinâmica

EM MUITAS APLICAÇÕES computacionais, o uso de variáveis estáticas é suficiente para resolução dos problemas. Porém, há situações que requerem uma otimização do uso da memória do computador. Imagine que você esteja jogando em seu computador, um jogo complexo, cheio de ação e com diversos níveis de duração. Dependendo da resolução da imagem e dos canais de som, seu tamanho poderá passar facilmente dos 5 gigabytes. Pensando nisso, o jogo precisa ser carregado aos poucos para a memória para ser exibido. Alguns níveis são carregados e, conforme o tempo vai decorrendo, a memória vai sendo liberada para dar espaço aos próximos níveis de jogo, e assim sucessivamente. Por este e diversos outros motivos, surge a necessidade de se alocar a memória do computador de forma dinâmica, isto é, com base na necessidade e disponibilidade. Neste capítulo

vamos saber como manipular os endereços de memória do computador em nossos algoritmos.

Objetivos de aprendizagem:

1. conhecer sobre alocação dinâmica da memória;
2. conhecer os comandos para manusear ponteiros;
3. conhecer as estruturas pilha e fila de forma dinâmica.

4.1 Alocação dinâmica de memória

Sabemos que quando é declarada uma variável estática em um programa, um espaço de memória é reservado para armazenar o dado desejado. A este espaço é vinculado um nome, denominado identificador (o nome que damos às variáveis no algoritmo), e a variável irá ocupar aquele espaço de memória durante toda a execução do algoritmo no qual ela foi criada. Durante a execução de um programa pelo computador, o espaço de memória utilizado para guardar o número inteiro poderia ser referenciado pelo identificador da variável, “num”, por exemplo. Mesmo utilizando essa forma de acesso pelo identificador, a variável foi alocada em algum “lugar” da memória: que chamamos de endereço de memória.

A partir do momento que podemos acessar e manusear o endereço de memória do computador em nossos algoritmos, podemos alocar dinamicamente os espaços de memória que iremos necessitar nos algoritmos.

A alocação dinâmica só é possível mediante uma mudança de paradigma quanto à manipulação de dados em memória: ao invés de se declarar variáveis de forma estática e acessá-las pelo seu identificador, faz-se necessário acessar determinada posição de memória diretamente pelo seu endereço. Para que uma variável possa ser manipulada pelo seu endereço de memória, ele precisa estar armazenado, referenciado, em outra variável. Assim como uma variável do tipo “num” armazena números inteiros, existe um tipo específico de variável que armazena endereços de memória: o ponteiro.

A figura 4.1 mostra a relação entre o conteúdo de uma variável e o endereço que a variável ocupa na memória do computador.

Figura 4.1 – Conteúdo e o endereço de memória em que ele está armazenado

Conteúdo	Endereço
0000 0001	0x0023FF16
0001 1001	0x0023FF17
0101 1010	0x0023FF18
1111 0101	0x0023FF19
1011 0011	0x0023FF1A

Fonte: Elaborada pelo autor.

4.2 Ponteiros

Quando declaramos uma variável, definimos um nome (identificador da variável) e então toda referência feita a esta variável é realizada por meio deste nome. Outra forma de referenciar uma variável é por meio de seu endereço. Por endereço entende-se o número que a variável ocupa na memória do computador.

As variáveis declaradas na área de declaração de variáveis são chamadas de variáveis estáticas e existem (são alocadas na memória) apenas durante a execução do bloco que as contém. Variáveis dinâmicas não são declaradas na área de declaração de variáveis, logo não podem ser referenciadas diretamente (não possuem identificadores). Para fazermos referência a uma variável dinâmica, usaremos o conceito de ponteiros. Os ponteiros são variáveis cujo conteúdo só pode ser o endereço (número) de uma variável na memória.

Para declarar um ponteiro, é preciso fornecer o tipo de variável para a qual ele irá apontar, isto é, p é um ponteiro do tipo t se houver a declaração:

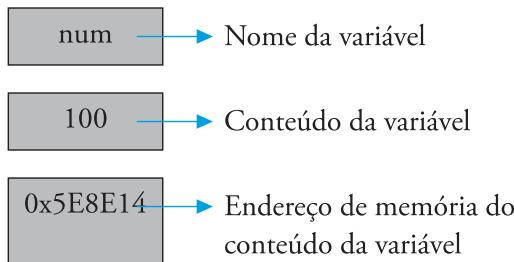
$$p = \text{PONTEIRO para } t;$$

onde t deve ser um tipo de dado já existente.

Dizer que p aponta para uma variável do tipo t , significa dizer que a variável p está armazenando o endereço de uma variável do tipo t , isso é, o ponteiro aponta para o primeiro endereço de uma área de memória reservada para armazenar o conteúdo da variável. A figura 4.2 mostra a

variável de nome NUM que tem o conteúdo 100, e este conteúdo está armazenado no endereço 0x5E8E14 da memória do computador, nesta execução do algoritmo.

Figura 4.2 – Variável, conteúdo e endereço de memória



Fonte: Elaborada pelo autor.

4.3 Comandos para manipulação de ponteiros

Para manipular endereços de memória, são necessárias algumas funções que estão disponíveis na maioria das linguagens de programação (C e Java, por exemplo) e permitem alocar e desalocar o endereço de memória utilizado.

- × ALOQUE (*p*): esta função reserva um espaço na memória, com tamanho suficiente para armazenar os dados pertencentes a uma variável do tipo nó (*p* é um ponteiro para o tipo nó).
- × DESALOQUE (*p*): esta função libera o espaço que foi alocado pela variável dinâmica (nó). Outra consequência deste comando é que o conteúdo da variável apontada passa a ser indefinido, isto é, *p* passa a apontar para um local de informações nulas.
- × *p* NIL: esta atribuição é **bastante importante quando manuseamos ponteiros** e faz com que a variável apontador *p* receba um valor indefinido, isto é, que a variável aponte para um local de informações nulas (NIL).

Os operadores que podem ser utilizados nas variáveis do tipo ponteiro são o * e o &.

- × `*`: utilizado para declarar uma variável do tipo ponteiro e retornar o valor contido no endereço de memória.
- × `&`: utilizado para retornar o endereço de memória que a variável está ocupando.

Exemplo 4.1

```

...
x: inteiro;
*ap : ponteiro para inteiro;           //apontador para inteiro
x <- 5;                                // coloca 5 na variável X
ap <- &x;                            // ap aponta para o endereço de X
imprima (*ap);                      // imprime o valor que está no endereço
apontado por ap, no caso 5

```

4.4 Utilização de ponteiros

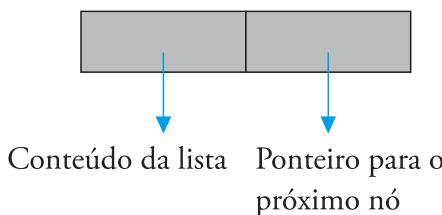
Nas estruturas de dados do tipo vetor e matriz, o número máximo de elementos deve ser conhecido ou estimado durante a construção do algoritmo, levando o programador a alocar uma quantidade de memória nem sempre necessária ou disponível. Este problema pode ser resolvido utilizando estruturas de alocação dinâmica, pois o número de endereços de memória alocados pode aumentar o diminuir em tempo de execução, permitindo uma melhor utilização da “memória de dados”. Outra característica importante da estrutura de alocação dinâmica é a facilidade de remoção e inserção de elementos, diferentemente das estruturas do tipo vetor e matriz, que podem exigir considerável movimentação dos dados neste tipo de operação.

4.5 Estrutura nó ou nodo

Quando estudamos as pilhas e filas (capítulos 2 e 3, respectivamente), utilizamos estas estruturas de forma estática, mas elas podem ser utilizadas de forma dinâmica. Isto é, ao invés de termos pilhas e filas com tamanhos estáticos em vetores, podemos utilizá-las de forma dinâmica, aumentando e diminuindo estas estruturas conforme a necessidade do algoritmo. Para isto precisamos utilizar uma estrutura chamada *nó* ou *nodo*.

A estrutura nó ou nodo compreende uma estrutura de dados do tipo registro, formada pelos campos que contêm o conteúdo propriamente dito do dado ou informação e o campo que contém os endereços de memória (ponteiros) dos nós subsequentes, para que se consiga ligar os componentes de uma pilha ou fila, por exemplo. A figura 4.3 mostra a estrutura nó.

Figura 4.3 - Estrutura de dados registro referente a um nó ou nodo da lista encadeada



Fonte: Elaborada pelo autor.

Para definirmos o registro da estrutura da figura 4.3 nós podemos utilizar o exemplo a seguir.

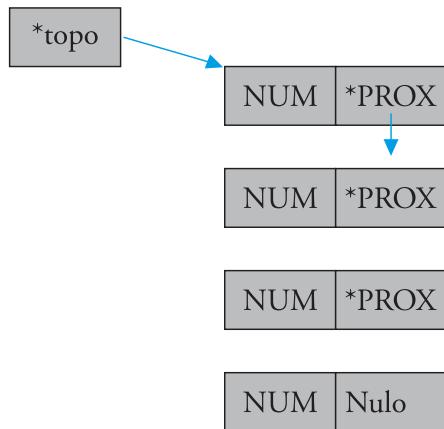
Exemplo 4.2

```
Tipo No = registro NUM: inteiro;  
                      *PROX: ponteiro para No;  
fimNo;
```

4.6 Estruturas dinâmicas pilha e fila

A estrutura pilha obedece ao seguinte comportamento: um elemento é inserido sempre no topo da pilha e um elemento é removido sempre do topo da pilha. A figura 4.4 mostra uma estrutura dinâmica referente a uma pilha, em que é preciso indicar o endereço do primeiro nó da pilha, o topo da pilha. O topo da pilha é uma variável do tipo ponteiro que aponta para o endereço do primeiro nó da pilha. Os registros que formam a pilha são compostos por dado e ponteiro para o próximo nó da pilha. O último nó da pilha contém seu nó, o campo dado e o campo ponteiro com o valor nulo. O valor nulo no ponteiro do nó indica final da pilha.

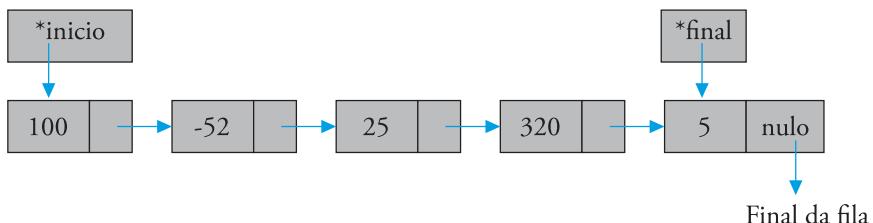
Figura 4.4 – Estrutura dinâmica da pilha



Fonte: Elaborada pelo autor.

A estrutura fila segue o seguinte comportamento: um elemento é sempre inserido no final da fila e um elemento é sempre removido do início da fila. Na figura 4.5 podemos ver uma estrutura dinâmica referente a uma fila, em que é preciso indicar o endereço do primeiro nó da fila, o início da fila. O início da fila é uma variável do tipo ponteiro que aponta para o endereço do primeiro nó da fila. Os registros que formam a fila também são compostos por dado e ponteiro para o próximo nó da fila. O último nó da fila contém em seu nó o campo dado e o campo ponteiro com o valor nulo. O valor nulo no ponteiro do nó indica final da fila. No caso da fila, fazemos a inserção de um nó sempre no final da fila, então precisamos ter um ponteiro para o final da fila também.

Figura 4.5 – Estrutura dinâmica da fila



Fonte: Elaborada pelo autor.

Para podermos manipular estas estruturas dinâmicas de pilha e fila sempre que formos inserir um nó, no topo da pilha ou no final da fila, precisamos antes alocar um endereço de memória do tipo registro NO e ligar este novo nó na estrutura. Quando precisarmos excluir um nó, do topo da pilha ou do início da fila, precisamos desligar o registro No da estrutura e desalocar o endereço de memória, liberando-o para outro uso.

Também é preciso atualizar os ponteiros topo (na pilha) e início e final (na fila), para que continuem apontando adequadamente para os endereços de topo da pilha e início e final da fila. Lembrando que as atividades de inserção e remoção de elementos na pilha e fila seguem a lógica da respectiva estrutura.

É interessante notar as seguintes situações:

1. quando o topo da pilha contiver o valor nulo, significa que a pilha está vazia;
2. quando os ponteiros início e final da fila contiverem o valor nulo, significa que a fila está vazia;
3. quando os ponteiros início e final da fila contiverem o mesmo valor, e este for diferente de nulo, indica que a fila contém somente um elemento;
4. quando as pilhas e filas são dinâmicas, não precisamos testar a estrutura cheia, pois vamos alocando na necessidade. A menos que tenhamos ocupado todo o espaço de memória do computador.

4.7 Algoritmos de manipulação de pilhas e filas

Na sequência serão apresentadas propostas de algoritmos de inserção e remoção de nó nas estruturas dinâmicas de pilha e fila.

A estrutura Nó utilizada nos algoritmos segue a definição a seguir:

```
Tipo NO = registro NUM: inteiro;
          *PROX: ponteiro para NO;
fimNO;
*TOPO, *INICIO, *FINAL: ponteiro para NO;
```

4.7.1 Manipulação de pilhas

Os algoritmos mostram propostas de empilhar (*push*) e desempilhar (*pop*) de nós na estrutura pilha. Lembrando que na pilha os nós são inseridos e removidos de seu topo (ver capítulo 2).

```

procedimento empilhar(*TP: TOPO)
início
    ATUAL: NO;
    ALOQUE (&ATUAL);
    Imprima ("Entre com um número: ");
    Leia (NO.NUM);
    se *TP = nulo
        então imprima("PILHA VAZIA, inserindo 1º valor!");
        TP ← &ATUAL;
        NO.*PROX <- nulo;
    Senão
        ATUAL ← *TP;
        NO.*PROX ← ATUAL.*PROX;
        *TP ← &NO;
    Fimse;
fim; // procedimento empilhar

procedimento desempilhar(*TP : TOPO)
início
    ATUAL: NO;
    se *TP = nulo
        então
            imprima("PILHA VAZIA, não existe valor para remover!");
            desempilhar;
    Senão
        ATUAL ← *TP;
        Imprima("Valor a ser desempilhado ", ATUAL.NUM);
        TP ← ATUAL.*PROX;
        Desaloque (&ATUAL);
    Fimse;
Fim; // procedimento desempilhar

```

4.7.2 Manipulação de filas

Os algoritmos mostram propostas de enfileirar (*enqueue*) e desenfileirar (*dequeue*) de nós na estrutura fila. Lembrando que na fila os nós são inseridos no final e removidos do início (ver capítulo 3).

Estrutura de Dados

```
procedimento enfileirar (*IN: INICIO, *FN: FINAL)
    inicio
        VALOR : inteiro;
        ATUAL, NOVO: NO;
        Aloque (&NOVO);
        Imprima ("Entre com um número:");
        Leia (NOVO.NUM);
        NOVO.*PROX ← nulo;
        se *FN = nulo
            então
                imprima("FILA VAZIA, inserir 1º valor!");
                *FN ← &NOVO;
                *IN ← &NOVO;
            Senão
                ATUAL ← *FN;
                ATUAL.*PROX ← &NOVO;
                *FN ← &NOVO;
        Fimse;
    Fim; // procedimento enfileirar
```

```
procedimento desenfileirar(*IN: INICIO,*FN: FINAL)
    inicio
        ATUAL: NO;
        se *IN = nulo
            então
                imprima("FILA VAZIA! Não existe valor para desenfileirar");
                desenfileirar;
            senão
                se *IN = *FN
                    então imprima ("existe somente um valor na fila");
                        ATUAL ← *IN;
                        imprima ("desenfileirando valor ", ATUAL.NUM);
                        desaloque (&ATUAL);
                        *FN ← nulo;
                        *IN ← nulo;
                    senão ATUAL ← &IN;
                        imprima ("desenfileirando valor ", ATUAL.NUM);
                        *IN ← ATUAL.*PROX;
                        Desaloque (&ATUAL);

                Fimse;
            Fimse;
    Fim; //procedimento desenfileirar
```

Síntese

Foram apresentados neste capítulo os recursos disponibilizados nas linguagens de programação para alocação dinâmica de memória. Este recurso permite que seja possível utilizar o espaço de memória na medida da necessidade dos algoritmos, sem desperdiçar espaço alocando-os previamente. A programação fica mais eficiente com a utilização destes recursos. Buscando a melhoria dos algoritmos, também foram apresentadas aqui as estruturas de dados pilha e fila com alocação dinâmica, melhorando a performance destas estruturas.

Atividades

- Assumindo que queremos ler o valor de X e o endereço de X foi atribuído a PTRX, a instrução seguinte é correta? Justifique.

Leia (*PTRX);

- Seja o seguinte trecho de algoritmo:

```
Início
    P, Q = ponteiro para inteiro;
    I, J : inteiro;
    I ← 3;
    J ← 5;
    P ← &I;
    Q ← &J;
    ....
```

Qual é o valor das seguintes expressões?

- $*P - *Q$
 - $3 - *P / (*Q + 7)$
- Qual será a saída deste algoritmo, supondo que I ocupa o endereço 4094 na memória?

```
Início
    P = ponteiro para inteiro;
    I : inteiro;
    I ← 5;
    P ← &I;
```

Estrutura de Dados

```
    Imprima (P, *P + 2, **&P, 3 * (*P), (**&P) + 4);  
Fim.
```

3. Se I e J são variáveis inteiros e P e Q ponteiros para inteiro, quais das seguintes expressões de atribuição são ilegais?
- a) $P \leftarrow \&I;$
 - b) $P \leftarrow \&\&I;$
 - c) $I \leftarrow (*\&J);$
 - d) $*\&J;$
 - e) $I \leftarrow *\&*\&J;$
 - f) $I \leftarrow (*P) + (3 * (*Q));$

5

Estrutura de Dados Lista

UMA LISTA FUNCIONA como um vetor, isto é, podemos inserir um novo elemento ou excluir um elemento já existente em qualquer posição que esteja disponível. Naturalmente, a inserção deve obedecer a um critério determinado. Ninguém gostaria de utilizar uma lista telefônica ou consultar um dicionário que não estivesse em ordem alfabética, certo?

Objetivos de aprendizagem:

1. entender as listas lineares;
2. conhecer as listas estáticas, seus algoritmos e aplicações;
3. conhecer as listas dinâmicas, seus algoritmos e aplicações.

5.1 Listas lineares

Uma lista linear é basicamente um vetor em que os elementos são de um mesmo tipo de dado (inteiro, real, caractere ou lógico) e estão organizados de uma maneira sequencial. A ideia de uma lista linear é que exista uma ordem lógica entre os elementos que compõem esta lista. Desta forma, uma lista linear permite que um conjunto de dados preserve uma relação de ordem entre seus elementos. Por exemplo: no atendimento de pacientes em um consultório odontológico, os pacientes sentam-se aleatoriamente na sala de espera, mas a ordem de atendimento segue uma organização prévia e todos sabem quem será o próximo paciente a ser atendido. Outros exemplos são: a lista de letras que forma uma palavra; a pilha de cartas de um baralho; a fila de atendimento no caixa do supermercado; entre outras situações.

Em computação, a alocação de memória do computador quando armazenamos os elementos de uma lista pode acontecer de duas formas: sequencial ou contígua; e encadeada. Em uma lista linear sequencial ou contígua, os elementos além de estarem em uma ordem lógica previamente descrita, também estão fisicamente em sequência. E a forma mais comum de armazenarmos uma lista linear sequencial é utilizando a estrutura que conhecemos como vetor (ver capítulo 1). Já nas listas lineares encadeadas, mesmo mantendo-se a ordem lógica de seus elementos, eles não precisam estar armazenados sequencialmente na memória. Veremos este caso neste capítulo.

Podemos realizar diversas operações com as listas lineares, quer elas sejam sequenciais ou encadeadas, como: criar uma lista, inserir elementos em uma lista, remover elementos da lista, acessar um elemento da lista, imprimir os elementos da lista, entre outros, porém sempre respeitando a ordem lógica de seus elementos.

As listas lineares podem ser do tipo pilha, cuja ordem lógica de seus elementos é o último elemento que entrou ser o primeiro a sair (ver capítulo 2);

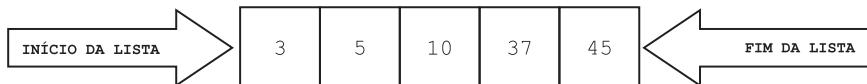
fila, cuja ordem lógica de seus elementos é o primeiro elemento a entrar ser o primeiro a sair (ver capítulo 3); e as listas lineares, cuja ordem lógica de seus elementos permite que eles entram e saiam pelo “*inicio*” ou pelo “*final*”. Este último tipo de lista também é conhecido como deque (*Double-Ended Queue*).

Percebemos então que o que diferencia os tipos de lista é basicamente a forma como as operações são realizadas sobre a lista, obedecendo a ordem lógica de seus elementos. Estas listas podem ser implementadas tanto de forma sequencial (em um vetor) como de forma encadeada, utilizando alocação de memória por meio do uso de ponteiros (ver capítulo 4).

5.2 Listas lineares estáticas

Para representar as listas lineares estáticas podemos utilizar um vetor, mas indicando uma organização de seu conteúdo, conforme a figura 5.1. Esta figura representa uma lista de números inteiros ordenados indicando o *início* e o *fim* da lista.

Figura 5.1 – Representação de uma estrutura de dados elementar tipo lista



Fonte: Elaborada pelo autor.

As operações que iremos realizar nesta lista devem manter a ordem lógica estabelecida, neste caso o conteúdo está em ordem crescente. Assim, as operações realizadas na lista devem manter esta ordem.

5.2.1 Inserção em uma lista estática

Para inserirmos um elemento em uma lista estática precisamos obedecer sua ordem lógica. Segue o algoritmo de inserção de elemento em uma lista linear estática, cujos elementos estão em ordem crescente.

```

procedimento inserir (V:VET)
início
VALOR, INDEX, J : inteiro;
se ELEMENTOS = MAX
então imprima("LISTA CHEIA!");

```

Estrutura de Dados

```
    inserir;
fimse;
imprima ("ENTRE COM UM NÚMERO : ");
leia (VALOR);
para INDEX de 1 até ELEMENTOS faça
se V[INDEX] > VALOR
então interrompa //para
fimse;
fimpara;
se ELEMENTOS > INDEX
então
    para J de ELEMENTOS até INDEX passo -1 faça
        V[J+1] ← V[J];
    Fimpara;
Fimse;
V[INDEX] ← VALOR;
ELEMENTOS ← ELEMENTOS + 1;
Fim; //procedimento inserir
```

5.2.2 Exclusão em uma lista estática

Quando excluímos um elemento em uma lista estática precisamos reorganizar seus elementos, para que não fique uma posição vazia na lista. Segue o algoritmo de exclusão de um elemento em uma lista linear estática, cujos elementos estão em ordem crescente.

```
procedimento excluir(V : VET)
início
VALOR, INDEX, J : inteiro;
se ELEMENTOS = 0
então imprima ("LISTA VAZIA!");
excluir;
fimse;
imprima ("ENTRE COM O ELEMENTO A SER EXCLUIDO : ");
leia (VALOR);
para INDEX de 1 até ELEMENTOS faça
se V[INDEX] = VALOR
então interrompa; //para
fimse;
fimpara;
```

```

se INDEX > ELEMENTOS
    então imprima("ELEMENTO NÃO LOCALIZADO");
    excluir;
fimse;
para J de INDEX até ELEMENTOS -1 faça
    V[INDEX] ← V[INDEX + 1];
fimpara;
ELEMENTOS ← ELEMENTOS - 1;
Fim; //procedimento excluir

```

5.2.3 Imprimir uma lista estática

Para imprimir os elementos de uma lista estática obedecendo sua organização lógica, precisamos realizar a impressão desde o início até o final da lista. Segue o algoritmo de impressão de uma lista linear estática, cujos elementos estão em ordem crescente.

```

procedimento imprimir_lista(V : VET)
início
J : inteiro;
imprima("INÍCIO DA LISTA : ");
se ELEMENTOS = 0
    então imprima("LISTA VAZIA!");
senão para J de 1 até ELEMENTOS faça
    imprima(V[J]);
fimpara;
fimse;
imprima("FIM DA LISTA");
fim; // procedimento imprimir_lista

```

Na sequência é apresentado o algoritmo completo com os procedimentos que manuseiam a estrutura lista linear estática.

```

algoritmo //LISTA para inteiros utilizando um vetor
CONST MAX = 5
CONST TRUE = 1
TIPO VET = VETOR[MAX] : inteiros;
ELEMENTOS : inteiro;
procedimento inserir(V : VET);
procedimento excluir(V : VET);
imprimir_lista(V : VET);

```

Estrutura de Dados

```
inicio
    LISTA : VET;
    OPÇÃO : inteiro;
    ELEMENTOS ← 0;
    repita
        imprima("1. INSERIR");
        imprima("2. EXCLUIR");
        imprima("3. SAIR");
        imprimir_lista(LISTA);
        imprima("ENTRE COM SUA OPÇÃO");
        leia(OPÇÃO);
        escolha(OPÇÃO)
            caso 1 : inserir(LISTA);
            fimcaso;
            caso 2 : excluir(LISTA);
            fimcaso;
            caso 3 : fimcaso; // algoritmo
        fimescolha;
        enquanto(TRUE) ;
    fim. //algoritmo

procedimento inserir(V : VET)
inicio
    VALOR, INDEX, J : inteiro;
    se ELEMENTOS = MAX
        então
            imprima("LISTA CHEIA!");
            inserir;
    fimse;
    imprima("ENTRE COM UM NÚMERO : ");
    leia(VALOR);
    para INDEX de 1 até ELEMENTOS faça
        se V[INDEX] > VALOR
            interrompa; //para
        fimse;
    fimpara;
    se ELEMENTOS > INDEX
        então
            para J de ELEMENTOS até INDEX passo -1 faça
                V[J+1] ← V[J];
            Fimpara;
    Fimse;
    V[INDEX] ← VALOR;
```

```

ELEMENTOS ← ELEMENTOS + 1;
Fim; //procedimento inserir

procedimento excluir(V : VET)
início
    VALOR, INDEX, J : inteiro;
    se ELEMENTOS = 0
        imprima("LISTA VAZIA!");
        excluir;
    fimse;
    imprima("ENTRE COM O ELEMENTO A SER EXCLUIDO : ");
    leia(VALOR);
    para INDEX de 1 até ELEMENTOS faça
        se V[INDEX] = VALOR)
            então
                interrompa; // para
            fimse;
        fimpara;
        se INDEX > ELEMENTOS
            então
                imprima("ELEMENTO NÃO LOCALIZADO");
                excluir;
            fimse;
        para J de INDEX até ELEMENTOS -1 faça
            V[INDEX] ← V[INDEX + 1];
        Fimpara;
        ELEMENTOS ← ELEMENTOS - 1;
Fim; //procedimento excluir

procedimento imprimir_lista(V : VET)
início
    J : inteiro;
    imprima("INÍCIO DA LISTA : ");
    se ELEMENTOS = 0
        então
            imprima("LISTA VAZIA!");
        senão
            para J de 1 até ELEMENTOS faça
                imprima(V[J]);
            fimpara;
    fimse;
    imprima("FIM DA LISTA");
fim; //procedimento imprimir_lista

```

5.3 Listas lineares encadeadas

Você já conhece as listas lineares estáticas, que são os vetores. Na declaração do vetor é preciso informar, entre colchetes, a quantidade de itens que a lista vai conter, e uma vez feito isto, o tamanho do vetor não pode mais ser alterado durante o algoritmo. Mas, e se durante a execução, forem necessárias mais posições do que se havia planejado? Ou ainda: se foi reservado um vetor para um número determinado de posições e utilizarmos menos do que o previsto? O uso de listas estáticas, apesar de facilitar bastante a implementação, não é otimizado.

Por isso a necessidade da estrutura de dados abstrata chamada lista dinâmica. Neste caso, o tamanho da lista não é fixo e, por meio da alocação de espaço de memória, é possível aumentar ou diminuir o tamanho da lista de forma dinâmica. Esta estrutura é chamada lista linear encadeada.

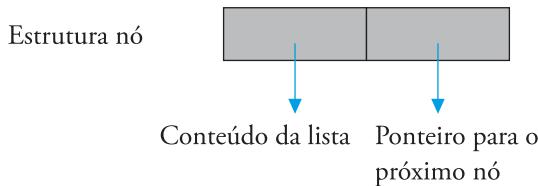
Em uma lista encadeada, ao invés dos itens serem armazenados contiguamente em memória e indexados, eles podem ser armazenados de forma dinâmica, em diferentes posições da memória. Porém, para que façam parte de uma única lista, e o acesso a todos os itens seja possível, eles precisam estar encadeados, ou seja, ligados uns aos outros de alguma forma. Daí que as listas encadeadas também são chamadas de listas ligadas.

Para armazenar os dados em uma lista encadeada, cada item da lista, ou seja, cada dado, é alocado em memória conforme a necessidade, e vinculado aos itens já existentes na lista. Utilizando este conceito, é possível inserir e retirar elementos da lista, alocando e liberando espaço, otimizando o uso da memória do computador.

5.3.1 Estrutura nó ou nodo

Antes de verificarmos os algoritmos para manipulação das listas lineares encadeadas, vamos rever uma estrutura nó ou nodo de uma lista. A estrutura nó compreende cada item da lista linear e corresponde a uma estrutura de dados registro, formada pelos campos que contêm o conteúdo propriamente dito da lista (o dado ou informação) e campos que contêm os endereços de memória (ponteiros) dos nós subsequentes, para que se consiga ligar os componentes da lista. Lembrando que uma lista encadeada também pode ser chamada de lista ligada. A figura 5.2 mostra a estrutura nó e a figura 5.3 mostra como estas estruturas se conectam para formar as listas encadeadas ou ligadas.

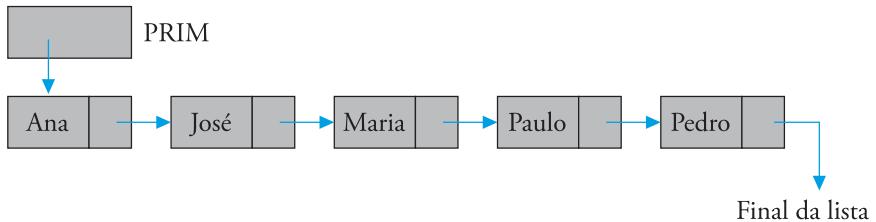
Figura 5.2 – Estrutura de dados registro referente a um nó ou nodo da lista encadeada



Fonte: Elaborada pelo autor.

No exemplo da figura 5.3, o único campo com dados é o nome do aluno que fará parte da lista, mas poderiam ser agrupados diversos outros campos, como por exemplo “matrícula”, “faltas” e “notas”, compondo um registro de notas do aluno da disciplina de Estrutura de Dados. Seja qual for o dado que está sendo cadastrado, além dos dados do usuário, o nó deverá conter o ponteiro para permitir o encadeamento, isto é, vincular este nó a outros nós existentes na lista. O último *nó* da lista, no campo *prox* conterá o valor Nulo, indicando que a lista terminou.

Figura 5.3 – Estrutura lista encadeada ou ligada e o endereço do primeiro nó da lista



Fonte: Elaborada pelo autor.

No exemplo da figura 5.3, o registro pode ser definido da seguinte forma:

```

Tipo No = registro NOME: caracter[40];
          *PROX: ponteiro para No;
fimNo;
*PRIM: ponteiro para NO;
```

Mas precisamos indicar para o algoritmo qual é o primeiro nó da lista, isto é, onde a lista inicia. Para isto, precisamos de uma variável do tipo pon-

teiro que armazene o endereço do primeiro nó da lista. Esta variável iremos chamar de *prim* (primeiro nó da lista), como é mostrado na figura 5.3. Quando o conteúdo de *prim* for nulo, isto indica que a lista está vazia, isto é, não aponta para nenhuma estrutura *nó* da lista.

Um nó pode ser inserido em qualquer posição de uma lista encadeada, bastando para isto atualizar os ponteiros indicadores das próximas posições, no caso do campo “*prox*”, tanto nos nós que já existem na lista quanto no nó que está sendo inserido.

5.3.2 Inserir dados na lista encadeada

Inserir um nó em uma lista encadeada é uma das possibilidades de manipulação de listas encadeadas. O primeiro passo para inserir um item na lista, é alocar um espaço de memória para armazenamento deste novo item. Neste caso, será alocado um espaço para armazenar um novo nó da estrutura. Sabendo-se o endereço de memória no qual este nó foi alocado, será possível preencher os campos do nó.

Mas é preciso verificar algumas situações de segurança quando alocamos um endereço de memória. Se a função malloc tentar fazer alocação de uma área de memória, mas por algum motivo ela não conseguir desempenhar esta tarefa junto ao sistema operacional, ao invés de ela retornar um endereço inicial da área que deveria ter sido alocada, ela retorna nula, indicando que não foi possível fazer a alocação. Então, antes de preencher os campos do item da lista, é preciso verificar se o nó foi de fato alocado.

Existem duas condições que precisamos considerar quando estamos inserindo um nó em uma lista encadeada: a lista está sem itens (vazia) e a lista já contém itens.

Caso a lista esteja sem itens, estamos inserindo o primeiro nó na lista. Neste caso o endereço do primeiro nó da lista tem o valor nulo (*prim* = null), então o endereço do nó que está sendo inserido será o primeiro endereço da lista (*prim*= endereço do primeiro nó da lista).

Caso a lista já contenha itens, o endereço do nó que está sendo inserido precisa ser vinculado ao último item da lista, sendo então ligado à lista. Para

inserir o novo nó na lista, precisamos obter o endereço do nó que acabou de ser alocado e atribuí-lo ao campo *prox* do último registro *no* atual da lista.

Porém, não existe nenhum ponteiro que guarde de forma estática o endereço do último elemento da lista. Temos o endereço do primeiro nó da lista (*prim*) e as ligações entre os nós que compõem a lista, por meio do campo *prox*, como se fosse uma corrente linear de pessoas de mãos dadas. Então, só conseguiremos acessar o último elemento da lista percorrendo toda a lista a partir do primeiro elemento, cujo endereço está no ponteiro estático *prim*.

Para isto será necessário um ponteiro auxiliar que inicia no endereço apontado por *prim* e, a partir dele, vai passando de nó em nó (campo *prox* de cada nó) até chegar ao final da lista, recebendo o endereço do último nó. Conhecendo o endereço do último nó, é possível fazer a atualização do ponteiro *prox* do último nó atual para o novo nó a ser inserido. E então o novo nó inserido será o último nó atual da lista.

O algoritmo a seguir mostra o processo de inserção mais comum (conforme descrito anteriormente), que é vincular o novo nó ao final da lista encadeada, ou seja, depois do último nó existente. O novo item é inserido sempre ao final da lista.

```
procedimento inserir (*PR:PRIM)
    inicio
        ATUAL, NOVO: NO;
        NOME: caracter[40];
        Aloque (&NOVO);
        imprima("ENTRE COM UM NOME : ");
        leia(NOME);
        se *PR = nulo
            então imprima("lista vazia, inserindo 1º nome");
            *PR ← &NOVO;
            NOVO.NOME ← NOME;
            NOVO.*PROX ← nulo;
        Senão ATUAL ← *PR;
            enquanto ATUAL.*PROX <> nulo faça
                ATUAL ← *(ATUAL.*PROX);
            Fim enquanto;
            imprima ("valor será inserido no final da lista");
            NOVO.NOME ← NOME;
            ATUAL.*PROX ← &NOVO;
            NOVO.*PROX ← nulo;
```

Estrutura de Dados

```
Fimse;  
Fim; //procedimento inserir
```

5.3.3 Buscar dado na lista encadeada

Outra atividade que podemos realizar na lista é buscar, pesquisar ou localizar um item específico. Uma das formas de se localizar um item em uma lista é por meio da pesquisa sequencial, que é a forma de pesquisa onde você parte do primeiro elemento de uma lista, e vai passando por cada um dos elementos, percorrendo a lista, até encontrar o item desejado. Esta atividade já realizamos no item 5.3.2, quando acessamos o último elemento da lista para encadeamento de um novo nó.

Na sequência será apresentado o algoritmo de uma função que solicita ao usuário um dado (número, nome, depende da informação que a lista está armazenando) para ser localizado na lista.

```
procedimento busca (*PR:PRIM, CHAVE:caracter[40])  
    inicio  
        ATUAL: NO;  
        se *PR = nulo  
            então imprima("lista vazia!");  
            busca;  
        senão ATUAL ← *PR;  
            enquanto ATUAL.*PROX <> nulo E ATUAL.NOME <> CHAVE faça  
                ATUAL ← *(ATUAL.*PROX);  
            Fim enquanto;  
            Se ATUAL.*PROX = nulo  
                Então imprima ("Nome não encontrado na lista");  
                Busca;  
            Senão imprima ("Nome encontrado no endereço ", &ATUAL,  
            "da lista");  
            Fimse;  
        Fimse;  
    Fim; //procedimento busca
```

5.3.4 Imprimir os dados da lista encadeada

Usando a mesma lógica da pesquisa sequencial, podemos listar todos os elementos de uma lista. Basta percorrer toda a lista e mostrar os valores dos campos de cada nó. Segue o algoritmo.

```

procedimento listar (*PR:PRIM)
    inicio
    ATUAL: NO;
    se *PR = nulo
        então imprima("lista vazia!");
        listar;
    senão ATUAL ← *PR;
        enquanto ATUAL.*PROX <> nulo faça
            imprima (ATUAL.NOME);
            ATUAL ← *(ATUAL.*PROX);
        Fim enquanto;
    Fimse;
Fim; //procedimento listar

```

5.3.5 Remover dado da lista encadeada

Da mesma forma que é possível alocar espaço de memória do computador de forma dinâmica, também é possível liberar a área de memória alocada. Quando removemos um item da lista encadeada também liberamos a área de memória ocupada pelo item removido. Porém, esta remoção precisa ser realizada com segurança.

Para remover um item da lista que está entre dois itens (no meio da lista, por exemplo), o item anterior ao item que será removido precisa apontar diretamente para o item posterior ao item removido. Então, basta fazer uma troca de endereço dos nós, isto é, trocar o conteúdo dos campos *prox* de cada nó envolvido, isolando o nó que será removido, e depois liberar o nó que foi removido.

Mas é preciso localizar o item a ser removido, e como não temos ponteiro estático para ele, precisamos utilizar a pesquisa sequencial, ou seja, a mesma estrutura repetitiva realizada no item 5.3.2, que inicia no *prim* da lista e vai verificando item por item da lista, por meio do campo *prox* de cada nó, até encontrar o nó que se deseja remover.

Outra situação de remoção é quando o nó a ser removido é o primeiro da lista. Neste caso, o ponteiro de início da lista, o *prim*, precisa ser alterado. Então, basta localizar o endereço do segundo nó da lista, fazer o *prim* apontar para este nó (o segundo nó da lista) e liberar o endereço do nó removido.

```

procedimento remover (*PR:PRIM)
    inicio

```

Estrutura de Dados

```
ATUAL, ANTERIOR: NO;
NOME: caracter[40];
Imprima ("digite nome a ser removido");
Leia (NOME);
se *PR = nulo
então imprima("lista vazia, não existe nome para remover");
    remover;
Senão ATUAL ← *PR;
enquanto ATUAL.*PROX <> nulo E ATUAL.NOME <> NOME faça
    ANTERIOR ← ATUAL;
    ATUAL ← *(ATUAL.*PROX);
Fim enquanto;
Se ATUAL.*PROX = nulo
    Então imprima ("Nome não encontrado na lista");
    remover;
Senão se &ATUAL = *PR
    Então imprima ("Nome encontrado e removido da 1ª
posição da lista");
        *PR ← ATUAL.*PROX;
        Desaloque (&ATUAL);
        Senão se ATUAL.*PROX = nulo
            Então imprima ("nome encontrado e removido da
última posição da lista");
                ANTERIOR.*PROX <- nulo;
                Desaloque (&ATUAL);
                Senão imprima ("nome encontrado e removido do
meio da lista");
                    ANTERIOR.*PROX <- ATUAL.*PROX;
                    Desaloque (&ATUAL);
                    Fimse;
                Fimse;
            Fimse;
        Fim;
//procedimento remover
```

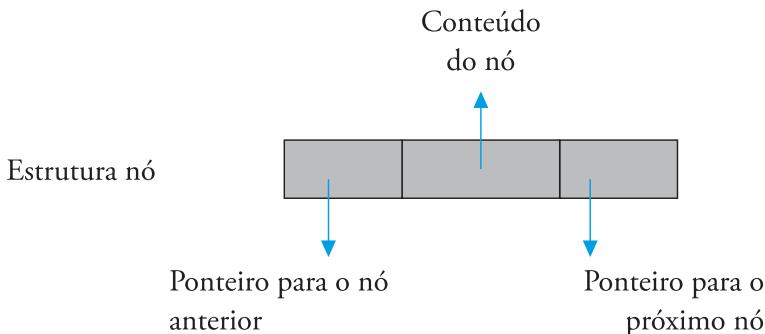
5.4 Listas lineares duplamente encadeadas

As listas encadeadas apresentadas no item 3 permitem que elas sejam percorridas somente do início (*prim*) até o final da lista. Da forma como os endereços dos nós estão armazenados, apontando para o próximo nó da lista (*prox*), não podemos percorrer a lista no sentido contrário. Isto significa que, se nossa lista de alunos da disciplina de Estrutura de Dados armazena os alu-

nos em ordem crescente de nome, teremos muito trabalho para imprimir esta lista em ordem decrescente de nomes (do Z até A). A proposta de uma lista linear duplamente encadeada pode resolver esta situação.

A lista duplamente encadeada tem dois ponteiros em cada nó, um que aponta para o próximo nó da lista (seu nó posterior) e outro ponteiro que aponta para o nó anterior a ele na lista (seu nó anterior). A figura 5.4 representa o nó de uma lista duplamente encadeada.

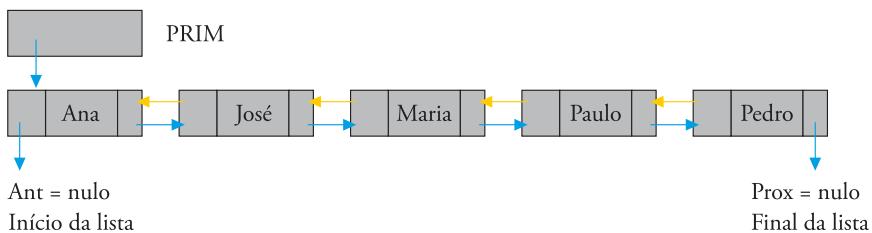
Figura 5.4 – Estrutura de dados registro referente a um nó da lista duplamente encadeada



Fonte: Elaborada pelo autor.

Na figura 5.5 está representada a lista duplamente encadeada que irá facilitar o manuseio da lista em qualquer sentido.

Figura 5.5 – Estrutura lista duplamente encadeada e o endereço do primeiro nó da lista



Fonte: Elaborada pelo autor.

No exemplo da figura 5.5, o registro pode ser definido da seguinte forma:

Estrutura de Dados

```
Tipo NO = registro *ANT: ponteiro para NO;
    NOME: caracter[40];
    *POS: ponteiro para NO;
fimNo;
*PRIM : ponteiro para NO;
```

5.4.1 Inserir dado na lista duplamente encadeada

A inserção de um nó em uma lista duplamente encadeada requer o controle dos dois ponteiros de cada nó – *ant* e *pos* –, para que a ligação entre os nós permaneça. Segue o algoritmo para inserção de um nó na lista duplamente encadeada.

```
procedimento inserir (*PR:PRIM)
    inicio
        ATUAL, NOVO, ANTERIOR: NO;
        NOME: caracter[40];
        Aloque (&NOVO);
        imprima("ENTRE COM UM NOME : ");
        leia(NOME);
        se *PR = nulo
            então imprima("lista vazia, inserindo 1º nome");
            *PR ← &NOVO;
            NOVO.NOME ← NOME;
            NOVO.*ANT ← nulo;
            NOVO.*POS ← nulo;
        Senão ATUAL ← *PR;
            enquanto ATUAL.*POS <> nulo faça
                ANTERIOR ← *(ATUAL.*ANT);
                ATUAL ← *(ATUAL.*POS);
            Fimenquanto;
            imprima ("valor será inserido no final da lista");
            NOVO.NOME ← NOME;
            ATUAL.*POS ← &NOVO;
            NOVO.*ANT ← &ATUAL;
            NOVO.*POS ← nulo;
        Fimse;
Fim; //procedimento inserir
```

5.4.2 Listar dados na lista duplamente encadeada

A partir de uma lista duplamente encadeada é possível listar os dados do início até o final da lista ou na ordem inversa, isto é, do final para o início da

lista. O grande benefício de uma lista duplamente encadeada é justamente a possibilidade de movimentação pelos nós da lista em qualquer sentido. Na sequência é apresentado o algoritmo.

```
procedimento listar_crescente (*PR:PRIM)
    início
    ATUAL: NO;
    se *PR = nulo
        então imprima("lista vazia!");
        listar;
    senão ATUAL ← *PR;
        enquanto ATUAL.*POS <> nulo faça
            imprima (ATUAL.NOME);
            ATUAL ← * (ATUAL.*POS);
        Fim enquanto;
    Fimse;
Fim //procedimento listar_crescente
```

```
procedimento listar_decrecente (*PR:PRIM)
    início
    ATUAL: NO;
    se *PR = nulo
        então imprima("lista vazia!");
        listar;
    senão ATUAL ← *PR;
        enquanto ATUAL.*POS <> nulo faça
            ATUAL ← * (ATUAL.*POS);
        Fim enquanto;
        enquanto ATUAL.*ANT <> nulo faça
            imprima (ATUAL.NOME);
            ATUAL ← * (ATUAL.*ANT);
        Fim enquanto;
    Fimse;
Fim; //procedimento listar_decrecente
```

5.4.3 Remover dado da lista duplamente encadeada

Para remover um nó de uma lista duplamente encadeada é preciso atentar para a correta alteração dos campos *ant* e *pos* dos nós anteriores e posteriores ao nó removido, para que a ligação entre os nós permaneça correta. Segue o algoritmo.

Estrutura de Dados

```
procedimento remover (*PR:PRIM)
início
ATUAL, ANTERIOR, AUX: NO;
NOME: caractere[40];
Imprima ("digite nome a ser removido");
Leia (NOME);
se *PR = nulo
então imprima("lista vazia, não existe nome para remover");
remover;
Senão ATUAL ← *PR;
enquanto ATUAL.*POS <> nulo E ATUAL.NOME <> NOME faça
    ANTERIOR ← ATUAL;
    ATUAL ← *(ATUAL.*POS);
Fim enquanto;
Se ATUAL.*POS = nulo
    Então imprima ("Nome não encontrado na lista");
    remover;
Senão se ATUAL.*ANT = nulo
    Então imprima ("Nome encontrado e removido da 1a
    posição da lista");
        AUX ← *(ATUAL.*POS);
        AUX.*ANT ← nulo;
        *PR ← ATUAL.*POS;
        Desaloque (&ATUAL);
    Senão se ATUAL.*POS = nulo
        Então imprima ("nome encontrado e removido da
        última posição da lista");
            ANTERIOR.*POS <- nulo;
            Desaloque (&ATUAL);
        Senão imprima ("nome encontrado e removido do
        meio da lista");
            ANTERIOR.*POS <- ATUAL.*POS;
            AUX ← *ATUAL.POS;
            AUX.*ANT ← &ANTERIOR;
            Desaloque (&ATUAL);
        Fimse;
    Fimse;
Fimse;
Fimse;
Fim;
//procedimento remover
```

5.5 Lista encadeada circular

Uma lista encadeada pode ser também circular. Sendo uma lista encadeada circular simples, ao invés do ponteiro *prox* do último nó da lista apontar para nulo (indicando final da lista), ele aponta para o endereço do primeiro nó da lista (ver figura 5.2).

Também podemos ter a lista duplamente encadeada circular, em que no último nó da lista o campo *pos* aponta para o primeiro nó da lista, e no primeiro nó da lista o campo *ant* aponta para o último nó da lista (ver figura 5.4), fechando o círculo.

5.6 Aplicações da estrutura de dados Lista

A estrutura de dados Lista mantém o conjunto de dados contidos nela em uma relação de ordem, crescente, decrescente ou até uma lista que contém filas e pilhas. Quando uma lista contém filas e pilhas, estas devem obedecer a sua forma de tratamento (ver capítulos 2 e 3). Assim uma Lista, além de envolver as operações básicas de inserção, remoção, busca e impressão de seu conteúdo, mantendo a relação de ordem definida inicialmente, também permite que se concatensem duas ou mais listas, que se informe o número de nós (itens) da lista, que se copie a lista, que se ordene de forma inversa o conteúdo da lista, e que se edite o conteúdo dos itens da lista, mantendo a relação de ordem. Estas outras operações podem ser realizadas pela combinação das operações básicas de inserção, remoção e busca.

Síntese

Neste capítulo buscou-se apresentar a estrutura de dados Lista, que em sua forma estática nada mais é do que um vetor. A partir do uso de alocação dinâmica, é possível melhorar significativamente o uso da estrutura de dados Lista, possibilitando aumentar e diminuir o tamanho da lista de acordo com a necessidade do momento do algoritmo que está sendo escrito, sem ocupar espaço desnecessário na memória do computador.

Também aproveitando o recurso de alocação dinâmica e da estrutura nó da Lista; a Lista encadeada, a Lista duplamente encadeada e a Lista duplamente encadeada circular são evoluções desta estrutura, que permitem a escrita de algoritmos eficientes para resolver situações complexas usando a computação. Os algoritmos básicos de manipulação das estruturas de Lista apresentados aqui e a combinação deles mostram a eficiência de sua aplicação em soluções computacionais.

Atividades

Escrever os algoritmos de (1) inserir, (2) remover, (3) buscar e (4) imprimir para a lista duplamente encadeada circular.

6

Recursividade

NA PROGRAMAÇÃO DE computadores, podemos organizar os algoritmos na forma de blocos que desempenham tarefas específicas. Esta organização é importante para a qualidade do software que é produzido, pois conseguimos manter o foco do algoritmo em uma só tarefa, facilitando o entendimento e manutenção destes algoritmos, além de permitir seu reuso. Este conceito é conhecido como modularização.

ESTES BLOCOS PODEM chamar a si mesmos. Sim, um bloco de algoritmo pode chamar ele mesmo. É como se fossem as bonecas *matrioska*, aquelas bonecas russas de mesma aparência que podem ser encaixadas de forma que fiquem uma dentro da outra (ver figura 6.1). Neste capítulo você verá como isto é possível e aprenderá a escrever estes algoritmos.

Estrutura de Dados

Figura 6.1 – Bonecas *matrioska* e sua representação



Fonte: Shutterstock.com/Chay/Robyn Mackenzie.

Objetivos de aprendizagem:

1. conhecer os conceitos de bloco de programas;
2. conhecer procedimentos, funções e suas aplicações;
3. conhecer o conceito e aplicação de recursividade.

6.1 Bloco

O nosso conhecimento de bloco até agora está limitado a inserir as palavras *início* e *fim* para definir o algoritmo. Entretanto, o conceito de bloco tem uma amplitude maior, pois podemos ter diversos blocos (uns dentro dos outros) dentro de um algoritmo.

De acordo com o que aprendemos anteriormente, um bloco era formado por:

```
inicio
    <declaração de variáveis>;
    <comandos>;
fim.
```

Agora, de acordo com o que vimos, podemos ter blocos dentro de blocos:

```
inicio
    <declaração de variáveis>;
    inicio {segundo bloco}
```

```

<declaração de variáveis>; {segundo}
    <comandos>; {segundo}
fim; {segundo}
<comandos>;
fim.

```

Uma das vantagens é a possibilidade de definir variáveis fora do seu lugar original, principalmente quando se conhece a variável no meio do algoritmo. Exemplo: inverter a ordem de um vetor.

```

inicio
    N: inteiro;
    leia (N);
    inicio //procedimento inverte
        tipo V=VETOR [1:N] inteiro;
        VET : V;
        leia (VET);
        // inverta VET
        imprima (VET);
    fim; //procedimento inverte
fim.

```

Uma variável é válida dentro do bloco onde foi declarada, isto é, a variável tem validade a partir do *início* do bloco e termina no *fim* do bloco. Por isto são chamadas de variáveis locais ao bloco.

Quando existe um bloco dentro de outro, as variáveis do bloco externo são locais para ele e globais para o bloco interno. Já as variáveis do bloco interno são locais a ele e são desconhecidas pelo bloco externo.

Quando existe conflito de nomes (duas variáveis com mesmo nome, sendo uma local e outra global), vale a definição local, e a global deixa de poder ser acessada (embora continue existindo).

Exemplo:

```

inicio
    I,J:inteiro;
    NOME: caracter[40];
    inicio //procedimento 1
        K,L,M:real;
        NOME: caracter[60];
        SOMA:inteiro;
        leia (NOME);

```

```
K ← I + J;  
SOMA ← K + (I*J);  
fim; //procedimento 1  
imprima (NOME);  
fim.
```

6.2 Procedimento

Procedimento é um bloco com nome. Podemos chamá-lo em qualquer ponto do algoritmo.

Sintaxe:

```
procedimento <nome> < lista de parâmetros>;  
<especificação dos parâmetros>;  
inicio  
    <declaração de variáveis locais>;  
    <comandos>;  
fim;
```

A chamada de um procedimento é apenas a colocação de seu nome em um comando, como se fosse um novo comando.

Sintaxe:

```
<nome do procedimento> (<parâmetros necessários>);
```

Exemplo: seja um procedimento para achar e imprimir o maior número entre 3 valores fornecidos.

```
inicio  
    N1,N2,N3:inteiro;  
    procedimento ACHA (A, B, C) //encontra maior valor  
        A,B, C:inteiro;  
        inicio  
            se A>=B e A>=C  
                entao imprima (A);  
            fimse;  
            se B>A e B>=C  
                entao imprima (B);  
            fimse;  
            se C>A e C>B  
                entao imprima (C);
```

```

        fimse;
    fim; //ACHA
    leia (N1,N2,N3);
ACHA (N1,N2,N3);
    leia (N1,N2,N3);
ACHA (n1,n2,n3):
    leia (N1,N2,N3);
ACHA (N1,N2,N3);
fim.

```

No cabeçalho do procedimento, temos uma série de variáveis, que podem ser de entrada e de saída. Chamamos estas variáveis de Parâmetros *formais*. Já as variáveis usadas quando o procedimento é efetivamente chamado são denominadas Parâmetros *efetivos*.

Os parâmetros efetivos de entrada podem ser variáveis, constantes ou expressões. Já os parâmetros efetivos de saída devem ser variáveis.

Um procedimento pode chamar outros dentro dele – são os procedimentos aninhados.

6.3 Função

Uma função é um bloco que tem um nome e que devolve um resultado. Em outras palavras, quando não houver retorno de um resultado, temos um procedimento, e quando houver, temos uma função.

Sintaxe:

```

Função <nome> <lista de parâmetros>: <tipo de dado do
resultado>;
    <declaração dos parâmetros>;
    inicio
        <declaração de variáveis locais>;
        <comandos>;
        <nome da função> ← < resultado>;
    fim;

```

Toda função usada dentro de um programa deve ter sido definida antes de ser chamada.

Sintaxe:

<variável global> \leftarrow <nome da função (<parâmetros>)>;

Exemplo: seja uma função que calcule o teorema de Pitágoras.

```
inicio
    W1,W2,RESP:real;
    Funcao PITAG (A,B) : real;
        A,B:real;
        inicio
            X:real;
            X  $\leftarrow$  (A ** 2 ) + (B ** 2 );
            PITAG  $\leftarrow$  X ** 0.5;
        fim; //PITAG
        leia (W1,W2);
        RESP  $\leftarrow$  PITAG (W1,W2);
        imprima (RESP);
    fim.
```

6.4 Recursividade

Recursividade é um conceito matemático no qual a solução de um problema é expressa como uma combinação de soluções de problemas idênticos, porém menores. A solução do menor problema possível consiste na solução de um caso extremo, o mais simples, que são as premissas sobre as quais a solução recursiva é criada (KANTEK; DE BASSI, 2013).

A existência de solução para o caso mais simples é necessária para que as equações de definição recursiva possam ser resolvidas (KANTEK; DE BASSI, 2013).

Vamos utilizar um exemplo clássico, o cálculo do fatorial de um número N para exemplificar o uso de recursividade. Mas o que é o cálculo do fatorial de um número? Bom, o fatorial de um número inteiro não negativo N é dado pelo produto de todos os números inteiros entre 1 e N. O valor fatorial de um número é denotado pelo símbolo de exclamação (!).

Por exemplo, o valor fatorial de 4 (4!) é dado por:

$$\underline{1 * 2 * 3 * 4 = 24.}$$

Números inteiros entre 1 e 4

Isto é, $4! = 24$.

O algoritmo não recursivo para cálculo do fatorial de N pode ser:

```
Função FATORIAL (N) : inteiro // não recursivo
N : inteiro;
Início
    FATORIAL ← 1
    Enquanto N > 0 faça
        FATORIAL ← FATORIAL * N;
        N ← N -1;
    Fim enquanto;
Fim.
```

Na função fatorial de um número N inteiro positivo temos as seguintes equações recursivas:

1. $\text{fatorial}(0) = 1$, que será a condição de parada das chamadas recursivas;
2. $\text{fatorial}(N) = N * \text{Fatorial}(N-1)$, que será a condição das chamadas recursivas.

Observe na segunda equação que o fatorial de um número N é expresso em função do fatorial do seu predecessor. A primeira equação trata do caso mais simples.

Quando um algoritmo é chamado *iterativo*, ele requer a repetição explícita de um processo até que determinada condição seja satisfeita, então este algoritmo poderá fazer uso de recursão. Em termos de programação de computadores, recursividade é utilizada quando existem rotinas que chamam a si próprias no decorrer do seu código.

6.4.1 Algoritmos de exemplo

Na sequência serão apresentados três problemas clássicos da matemática, em que o uso de uma solução recursiva facilita o processamento. Atente para

Estrutura de Dados

a comparação entre os algoritmos recursivos e não recursivos como solução para o mesmo problema matemático apresentado.

1. Cálculo do fatorial de um número N utilizando uma função recursiva

Considerando que:

- × fatorial (0) = 1, condição de parada da chamada recursiva;
- × fatorial (N) = N * Fatorial (N-1), chamada recursiva.

Segue a função recursiva para cálculo do fatorial de N, onde em destaque estão as chamadas recursivas da função.

```
Função FATORIAL (N) : inteiro // recursivo
    N : inteiro;
    Início
        se N > 0
            Então FATORIAL ← N * FATORIAL (N -1) ;
            Senão FATORIAL ← 1 ;
        fimse;
    Fim.
```

A tabela a seguir representa o teste de mesa com os resultados de cada chamada recursiva da função *fatorial* descrita acima com N valendo 4. Perceba na tabela que as funções são chamadas uma a uma, e somente quando o teste de mesa chega na condição de parada é que cada chamada anterior à função fatorial retorna seu valor, uma a uma, e consequentemente o resultado é calculado, retornando o valor do fatorial.

Tabela 6.1 – Teste de mesa da função recursiva – 4!

Chamada recursiva	Retorno da chamada
FATORIAL (4)	24
4 * FATORIAL (3)	4 * 3 * 2 * 1 * 1
3 * FATORIAL (2)	3 * 2 * 1 * 1
2 * FATORIAL (1)	2 * 1 * 1
1 * FATORIAL (0)	1 * 1
1	1

Fonte: Elaborada pelo autor.

2. Cálculo do MDC entre M (dividendo) e N (divisor)

O que é mesmo o MDC entre M e N? MDC é o máximo divisor comum entre os valores de M e N, isto é, indica o maior valor que pode dividir M e N. Por exemplo, o maior valor que divide ao mesmo tempo 12 e 36 é 12, então $\text{MDC}(12, 36) = 12$. Da mesma forma, o maior valor que divide ao mesmo tempo 18 e 24 é 6, então $\text{MDC}(18, 24) = 6$. Mas para encontrar este valor é preciso realizar um processo de fatoração, então precisamos listar todos os divisores de M e N e verificar qual o maior valor que divide M e N ao mesmo tempo.

Em torno do ano 300 a.C., Euclides definiu o cálculo para identificar o MDC entre dois valores M e N sem a necessidade de fatoração. Segue o algoritmo Euclidiano para cálculo do MDC (M, N) (INFOESCOLA, 2016):

```

Função MDC (M,N): inteiro; // não recursivo
    M,N: inteiro;
    Início
        AUX: inteiro;
        se N > M
            então AUX ← M;
            M ← N;
            N ← AUX;
        Fimse;
        Enquanto N <> 0 faça
            AUX ← M mod N;
            M ← N;
            N ← AUX;
        Fim enquanto;
        MDC ← AUX;
    Fim.

```

A partir deste algoritmo foi possível calcular o MDC recursivamente usando o resto da divisão de M por N como entrada para o próximo passo.

Para podermos aplicar um algoritmo recursivo no cálculo do MDC, precisamos identificar a condição de parada e a condição de recursão. Com base no conceito do algoritmo de Euclides para o cálculo do MDC (M, N), temos que:

Estrutura de Dados

- × $\text{MDC}(\text{M}, 0) = \text{M}$. Esta será a condição de parada do processo recursivo;
- × $\text{MDC}(\text{M}, \text{N}) = \text{MDC}(\text{N}, (\text{M mod N}))$, para $\text{N} > 0$, e esta será a condição da recursão.

Considerando esta situação, temos que:

$$\text{MDC}(12, 18) = \text{MDC}(18, 12) = \text{MDC}(12, 6) = \text{MDC}(6, 0) = 6$$

Assim, segue a função recursiva para $\text{MDC}(\text{M}, \text{N})$, onde em destaque estão presentadas as chamadas recursivas da função.

```
Função MDC (M, N) : inteiro; //recursivo
    M, N: inteiro;
    Início
        se N > M
            então MDC ← MDC (N,M) ;
        senão se N = 0
            então MDC ← M;
            senão MDC ← MDC (N, (M mod N)) ;
        fimse;
    fimse;
Fim.
```

3. A Série de Fibonacci também é um exemplo clássico de aplicação de recursão. Vamos conhecê-la?

A série de números (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...) é conhecida como Série de Fibonacci. Fibonacci, cujo nome era Leonardo Pisa, foi um matemático do século XIII que propôs esta lei de formação simples que consiste no seguinte: cada elemento, a partir do terceiro, é obtido somando-se os dois anteriores. Então, iniciando-se dos valores 1, 1 temos que $1+1=2$, $1+2 = 3$, $2+3 = 5$, $3+5=8$ e assim sucessivamente (INFOESCOLA, 2016).

Segue o algoritmo não recursivo para calcular os N primeiros termos da Série de Fibonacci:

```
Função FIB (N): inteiro;
    N: inteiro;
    inicio
        I, ANT, PEN, ULT: inteiro;
        ANT ← 0;
```

```

PEN ← 1;
FIB ← 0;
para I de 1 até N faça
    ULT ← ANT + PEN;
    FIB ← FIB + ULT;
    ANT ← PEN;
    PEN ← ULT;
fimpara;
fim;

```

Novamente, para podermos criar um algoritmo recursivo para esta função, é preciso identificar o ponto de parada e o ponto de chamada recursiva. Analisando o algoritmo anterior, é possível identificar que o controle de parada é quando o algoritmo atinge o “N-ésimo” termo desejado da série, então considerando que N é a quantidade de valores desejados da série de Fibonacci:

- ✗ N = 0: Fib (0) = 0, indica a condição de parada da recursão.
- ✗ N = 1: Fib (1) = 1, indica a condição de parada da recursão.
- ✗ N > 1: Fib (N) = Fib (N-1) + Fib (N-2), indica a chamada recursiva.

Neste exemplo temos uma situação em que a chamada recursiva ocorre duas vezes (FIB (N-1) e FIB (N-2)), para que seja possível gerar a soma dos dois termos anteriores. Segue a função recursiva para cálculo dos N primeiros termos da Série de Fibonacci. Em destaque estão sinalizadas as chamadas recursivas.

```

função FIB (N) : inteiro; //recursivo
    N: inteiro;
    inicio
        se N<= 1
            então FIB ← 1;
            senão FIB ← FIB(N-1) + FIB(N-2);
        fimse;
    fim;

```

Outra forma de escrever o algoritmo recursivo, considerando os dois primeiros termos na série como 1, é a que segue.

```

função FIB (N) : inteiro; // recursivo
    N: inteiro;

```

Estrutura de Dados

```
inicio
    se N= 1 ou N=2
        então FIB ← 1;
        senão FIB ← FIB(N-1) + FIB(N-2);
    fimse;
fim;
```

Aproveite os algoritmos anteriores para realizar os testes de mesa e fixar o processo de chamada e retorno das funções recursivas. Esta é uma ótima forma de entender como funciona a recursividade.

A execução de um procedimento recursivo é geralmente efetuada pelo uso de sub-rotinas. Como uma rotina recursiva chama a si própria, é necessária uma área de memória para salvar o seu contexto antes dessa chamada recursiva (registro de ativação), a fim de que o valor de retorno da função, possa ser restaurado ao final dessa chamada. Esses registros de ativação são organizados em forma de Pilha (veja no capítulo 2).

6.4.2 Profundidade

Chamamos de profundidade o número de vezes que uma rotina recursiva chama a si própria, até obter o resultado.

O cálculo do fatorial de 4 possui profundidade 4, ou seja, quatro níveis de recursão:

$4! = 4 * 3!$	Nível 1
$3! = 3 * 2!$	Nível 2
$2! = 2 * 1!$	Nível 3
$1! = 1 * 0!$	Nível 4
$0! = 1$	Parada

O cálculo de MDC (12, 18) possui profundidade 3, ou seja, três níveis de recursão:

$MDC(12, 18) = MDC(18, 12)$	Nível 1
$MDC(12, MDC(18 \text{ mod } 12)) = MDC(12, 6)$	Nível 2

MDC (6, MDC (12 mod 6)) = MDC (6, 0)	Nível 3
MDC = 6	Parada

6.4.3 Aplicações

Na maioria dos casos não é preciso utilizar recursividade, pois de um modo geral todo programa recursivo pode ser escrito em forma não-recursiva; no entanto, em algumas situações ela é realmente oportuna.

A principal vantagem de sua utilização é a redução do código-fonte da rotina. Entretanto, o uso de recursividade apresenta algumas desvantagens, tais como:

- ✗ baixo desempenho na execução, devido ao tempo gasto no gerenciamento da pilha de registros de ativação (empilhamento e desempilhamento das chamadas de função) e o espaço por ela ocupado;
- ✗ dificuldade de depuração dos programas, particularmente se a recursão for muito profunda (veja na seção 6.4.2).

Os principais critérios a serem utilizados para decidir se a recursividade deve ou não ser utilizada é a clareza do algoritmo e o desempenho esperado pelo algoritmo.

Alguns exemplos de aplicações de algoritmos recursivos são em problemas envolvendo manipulações de estruturas de dados do tipo árvores, algoritmos que trabalham com analisadores léxicos recursivos de compiladores e problemas envolvendo tentativa e erro, os chamados *backtracking*. Também as linguagens de programação voltadas para programação lógica e programação funcional utilizam a recursão como única estrutura de repetição, substituindo os comandos `para`, `enquanto` e `repita`, que não estão disponíveis nessas linguagens de programação.

Síntese

Neste capítulo foi apresentada a modularização, que consiste em separar os algoritmos em módulos do tipo função e procedimento para facilitar a compreensão, reuso e alteração dos algoritmos quando necessário. A partir do

conceito de módulos é possível conhecer e utilizar a recursividade em programação de computadores, que torna os algoritmos menores e mais elegantes, mas muitas vezes a lógica fica mais complexa.

Atividades

1. Implementar um algoritmo recursivo que realize a operação de potência (N^E , em que N é a base e E é o expoente) sem utilizar o comando `**`. Exemplo: 2^4 indica que a base 2 é multiplicada 4 vezes, isto é, $2 * 2 * 2 * 2 = 16$. Não esqueça que, por definição matemática, $N^0 = 1$.
2. Implementar um algoritmo recursivo que realize a operação de multiplicação ($X * Y$) sem utilizar o comando `*`. Exemplo: $2 * 4$ indica que o 2 é somado 4 vezes, isto é, $2 + 2 + 2 + 2 = 8$. Não esqueça que, por definição matemática, $X * 0 = 0$.
3. Escreva os algoritmos não recursivos para resolver as atividades 1 e 2.
4. Considerando a função recursiva a seguir:

```
Função X (A) : inteiro;
    A: inteiro;
    Inicio
        Se A <= 0
            Então X ← 0;
            Senão X ← A + X (A - 1);
        Fimse;
    Fim.
```

- a) Indique o resultado do algoritmo quando A vale 5, 1 e 0;
- b) Escreva o algoritmo não recursivo para a função X e teste o resultado com A valendo 5, 1 e 0.

7

Árvores

NESTE CAPÍTULO IREMOS tratar da estrutura de dados árvore, que não obedece a uma estrutura linear e é uma das mais importantes estruturas de dados. A estrutura árvore combina as vantagens das estruturas de dados vetor ordenado e lista encadeada em uma só estrutura: permite a busca de elementos de forma rápida como em

um vetor ordenado; e a inserção e exclusão de itens também de forma rápida, como em uma lista encadeada.

A principal característica das árvores é que os itens que as compõem, chamados de nós, são dispostos de forma hierárquica, respeitando a topologia em árvore. Desta forma, ao invés dos nós se conectarem em sequência, eles aparecem acima ou abaixo dos outros elementos da árvore. A forma de organização desta estrutura permite inúmeras aplicações na solução de problemas computacionais.

Objetivos deste capítulo:

1. compreender o conceito da estrutura de dados árvore;
2. identificar os diferentes tipos de estruturas de dados árvore;
3. conhecer as aplicações e algoritmos de árvores.

7.1 Conceitos relacionados à estrutura de dados árvore

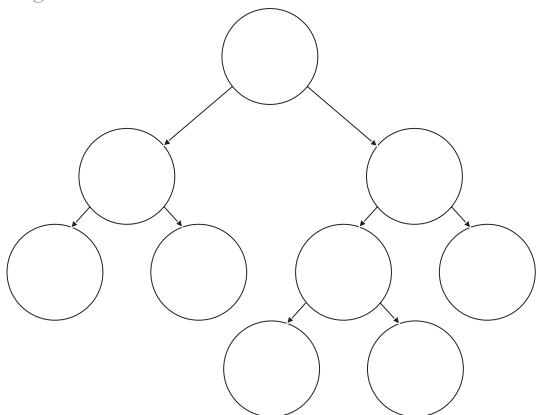
Na sequência serão apresentadas algumas definições formais de termos desta estrutura de dados, para que seja possível o entendimento e a implementação de árvores nos algoritmos. A maioria dos termos está relacionada com árvores reais (raiz, folha) e as relações familiares (pai, filho). Vamos iniciar pelos elementos que compõem a árvore.

Uma árvore consiste de **nós** conectados por bordas, conforme a figura 7.1. Os nós são representados por círculos e as **bordas** como linhas conectando os círculos.

O **nó raiz** de uma árvore é o nó que se encontra na região mais alta da árvore em relação aos outros nós. É possível perceber na figura 7.1 que a

árvore é representada de cabeça para baixo. Uma árvore contém somente um nó raiz e, para que um conjunto de nós e bordas seja considerado uma árvore, deve haver um percurso da raiz para qualquer outro nó. **Percorrer** uma árvore significa visitar todos os nós em alguma ordem especificada. Um nó é **visitado** quando o controle de um algoritmo chega no nó, geralmente com a finalidade de realizar algumas operações no nó.

Figura 7.1 – Árvore binária



Fonte: Elaborada pelo autor.

Qualquer nó, exceto a raiz, tem exatamente uma borda indo para cima até outro nó. O nó abaixo dele é chamado de **pai** do nó.

Qualquer nó pode ter uma ou mais linhas indo para baixo até outros nós. Estes nós abaixo de um nó são chamados de seus **nós filhos**.

Um nó que não tem filhos é chamado de um **nó folha**. Só pode haver uma raiz em uma árvore, mas pode haver muitas folhas.

Formalmente, uma árvore “A” é um conjunto finito de zero ou mais nós, tal que:

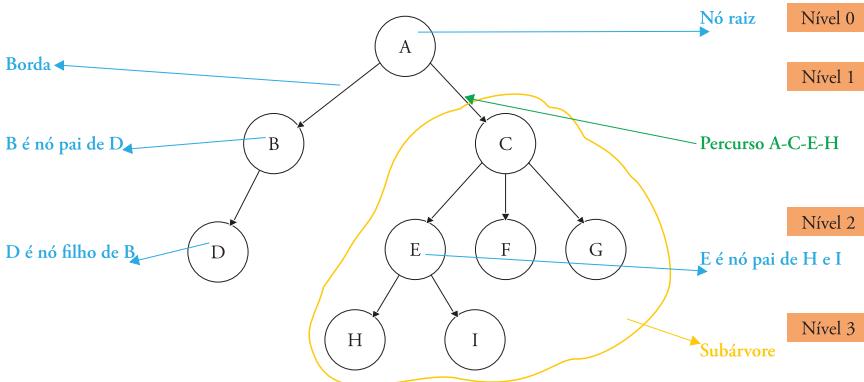
- ✗ se o número de nós for igual a zero, temos uma árvore vazia;
- ✗ se o número de nós for maior do que zero, os nós restantes formam conjuntos disjuntos, e cada um desses conjuntos é uma árvore por si só, chamada de **subárvore** da raiz de “A”.

O **nível** de um nó particular refere-se a quantas gerações o nó está da raiz. Considerando que a raiz está no nível zero, os seus filhos estarão no nível

Estrutura de Dados

1, seus netos no nível 2, e assim por diante. A figura 7.2 mostra os termos de uma árvore que foram definidos anteriormente.

Figura 7.2 – Conceitos e termos da estrutura árvore



Fonte: Elaborada pelo autor.

7.2 Representação de árvores

Em algoritmos, existem diversos tipos de árvores, classificadas de acordo com o percurso. Temos primeiramente as chamadas árvores genéricas, nas quais a hierarquia dos nós e a quantidade de níveis não obedecem a uma regra ou ordenação. Mas a maior parte das aplicações de árvores envolve as árvores ordenadas. Dentro deste contexto, as mais conhecidas são as árvores binárias de busca. Na sequência, comentaremos sobre os tipos mais utilizados de árvores.

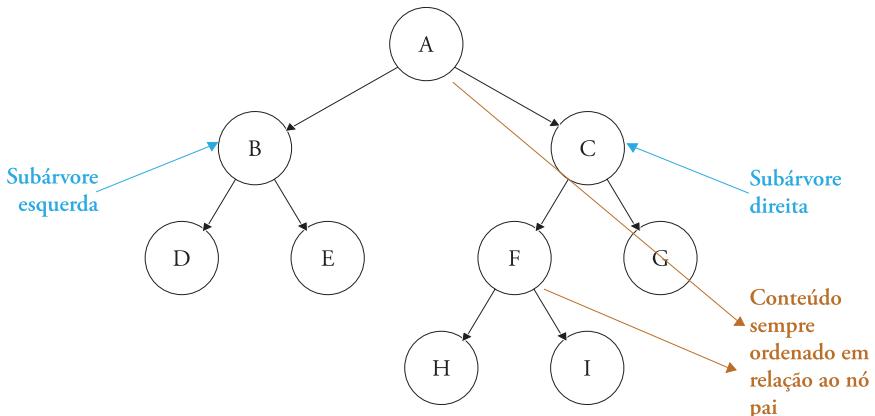
7.2.1 Árvore binária

Uma árvore binária é uma estrutura que, se não for vazia, possui um nó inicial, chamado raiz, com ponteiros para duas estruturas diferentes, denominadas **subárvore da esquerda** e **subárvore da direita**.

Uma árvore binária de busca é uma árvore binária que respeita uma regra para inserção de seus elementos. Desta forma, ela pode ser classificada como uma árvore ordenada. A figura 7.3 mostra uma árvore binária com suas

estruturas de subárvores direita e esquerda, a qual, como tem seus elementos ordenados, é considerada uma árvore de busca binária.

Figura 7.3 – Árvore binária de busca



Fonte: Elaborada pelo autor.

A aplicação mais comum de árvores binárias de busca é aquela em que, para cada nó, os filhos de menor valor são inseridos à esquerda, e os de igual ou maior valor são inseridos à direita. O respeito a esta regra permite que um elemento seja localizado nesta estrutura árvore quando solicitado.

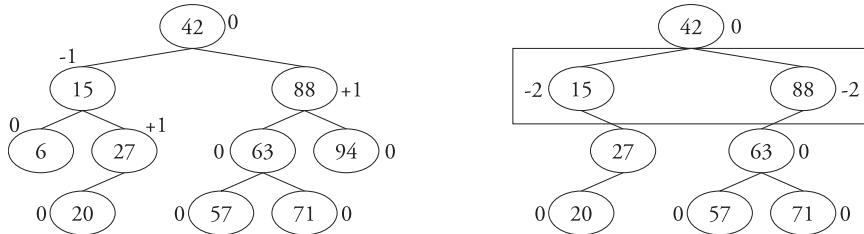
Além da implementação generalista de árvores binárias de busca, em que a única preocupação recai sobre respeitar a regra de inserção dos elementos para posterior recuperação, existem tipos de árvores binárias de busca que se preocupam também com o seu **balanceamento**.

7.2.2 Árvores AVL

Existem árvores binárias de busca que se preocupam com o balanceamento, e a árvore AVL é a precursora deste tipo de árvore. Proposta em 1962, a sigla é proveniente das iniciais dos sobrenomes dos seus inventores (Georgy Adelson-Velsky e Evgenii Landis). Ela é uma árvore balanceada pela altura, ou seja, a altura dos nós é tida como parâmetro para fazer o balanceamento. E o balanceamento ocorre por rotação, a fim de preservar a ordenação da árvore. A figura 7.4 mostra uma árvore AVL e uma não-AVL, por não estar平衡ada onde está sinalizado pelo quadrado.

Estrutura de Dados

Figura 7.4 – Árvore AVL

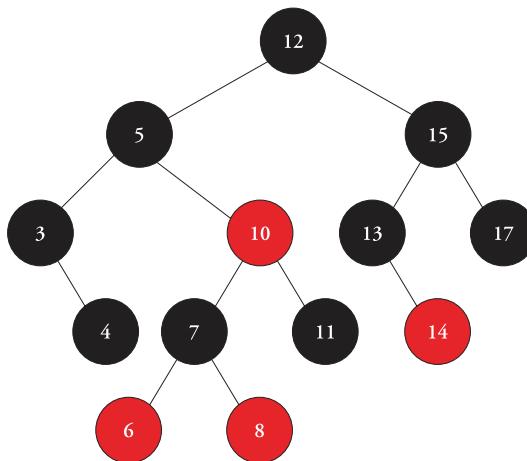


Fonte: Lafore (1999).

7.2.3 Árvore rubro-negra

Outro tipo de árvore平衡ada, e que muitas vezes é confundida com a AVL, é a rubro-negra. Este tipo de árvore foi proposto por Rudolf Bayer, em 1972, e se mostra um dos mais eficientes na inserção, busca e remoção de elementos. A árvore rubro-negra tem esse nome porque cada nó de sua estrutura possui um atributo de cor, vermelho ou preto, além de outras características. A figura 7.5 mostra uma árvore rubro-negra.

Figura 7.5 – Árvore rubro-negra

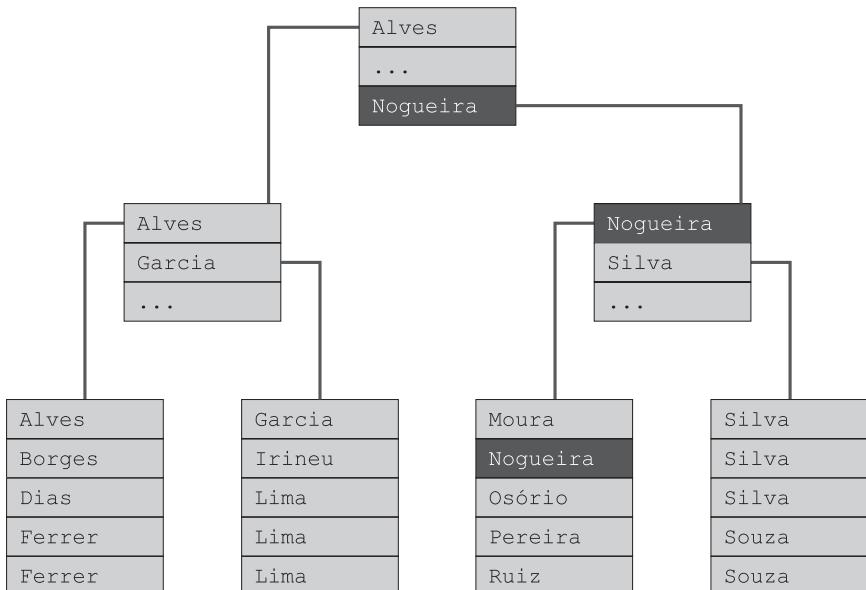


Fonte: Lafore (1999).

7.2.4 Outros tipos de árvores

Além da AVL e da rubro-negra, há outros tipos de árvores, como as árvores B e B+, projetadas para funcionar em memória secundária, como um disco rígido, por exemplo. Por conta disso, são bastante utilizadas em sistemas de banco de dados, como pode ser visto na figura 7.6.

Figura 7.6 – Árvore B

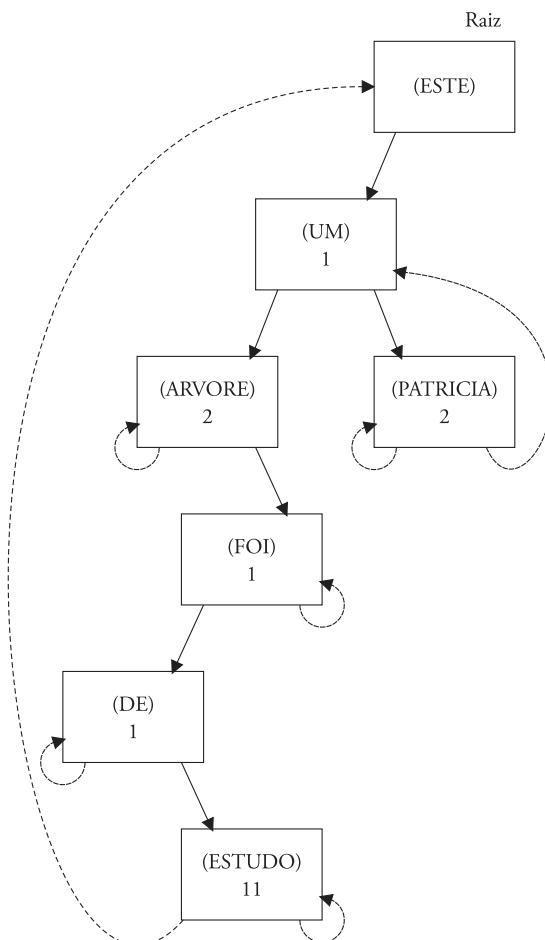


Fonte: <https://msdn.microsoft.com/pt-br/library/cc518031.aspx>

Existem ainda as árvores não binárias de busca, como a árvore patrícia, cuja principal aplicação é no tratamento de variáveis com conteúdo longo, como palavras e frases, como mostra a figura 7.7.

Estrutura de Dados

Figura 7.7 – Árvore patrícia



Fonte: <https://msdn.microsoft.com/pt-br/library/cc518031.aspx>

No contexto deste capítulo, nos concentraremos nas árvores binárias, uma vez que elas são as mais simples e mais comuns, além de serem as mais utilizadas. Os algoritmos apresentados na sequência serão focados na generalização da árvore binária de busca, tomando como foco principal a regra de inserção, e não o balanceamento ou a optimização.

Saiba mais

Seguem links de simulador de alguns tipos de árvore citados. Nestes simuladores você insere os nós na árvore, remove nós, busca um determinado nó e imprime a árvore. As movimentações são demonstradas de forma animada e sempre respeitando o conceito da árvore em questão.

No link <<https://www.cs.usfca.edu/~galles/visualization/BST.html>> você acessa um simulador para manipular árvores binárias de busca.

No link <<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>> você acessa um simulador para manipular árvores AVL.

No link <<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>> você pode visualizar o comportamento de árvores rubro-negras.

Árvore B no link <<https://www.cs.usfca.edu/~galles/visualization/BTree.html>> e Árvore B+ no link <<https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>>.

Aproveite! É uma forma divertida de entender o comportamento dos diversos tipos de árvore.

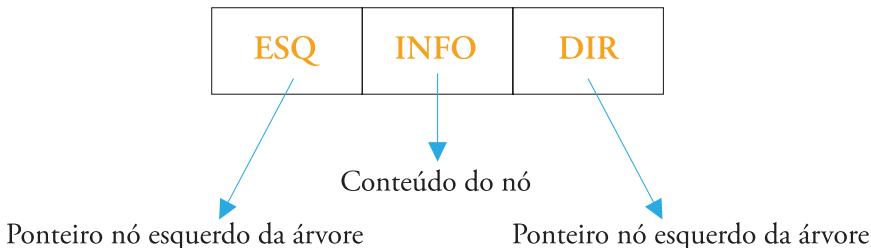
7.3 Algoritmos de manipulação de árvores

Agora que já conhecemos as estruturas de dados em forma de árvores, seus componentes e formas de manipulação, vamos entender a lógica de criação e uso destas estruturas nos algoritmos. Faremos a implementação de uma árvore binária de busca, porém sem nos preocuparmos com o balanceamento.

7.3.1 Criar uma estrutura árvore

A criação de uma árvore binária é feita por meio da alocação dinâmica de memória para comportar os nós, utilizando estruturas heterogêneas (os registros) e fazendo uso de ponteiros.

Figura 7.8 – Estrutura registro do nó da árvore



Fonte: Elaborada pelo autor.

```
Tipo REG = registro
    *ESQ: ponteiro para REG;
    INFO: inteiro;
    *DIR: ponteiro para REG;
fimREG;
*NO: REG;
```

7.3.2 Inserir elemento na árvore

Tendo uma estrutura de árvore binária de busca estruturada em memória, é possível inserir elementos nesta árvore. Uma árvore binária de busca tem como característica que seus elementos estão ordenados, então, quando inserimos um novo elemento na árvore, este deve ser inserido de forma a respeitar a ordem do conteúdo já existente. Por isto, é preciso pesquisar na árvore a posição onde o novo elemento será inserido.

A partir da raiz, percorre-se a árvore em níveis, obedecendo a regra de esquerda e direita, até encontrar um nó que tenha disponibilidade para receber o vínculo do novo nó que está sendo inserido. Para isto, é preciso alocar memória para o novo nó atribuindo *null* para os ponteiros esquerda e direita, já que todo elemento novo quando inserido se torna folha e não possui filhos. Este processo pode ser feito de forma iterativa ou de forma recursiva. Vamos apresentar o algoritmo de forma iterativa.

```
Procedimento Inserir (RAIZ: NO, VALOR: inteiro)
    inicio
    AUX, ANT: NO;
    Se RAIZ = null //primeiro nó da árvore
```

```

Então RAIZ.INFO ← VALOR;
RAIZ.DIR ← null;
RAIZ.ESQ ← null;
Senão AUX ← *RAIZ //árvore já existe
    ANT ← null
Enquanto AUX <> null faça //percorre a árvore
    ANT ← *AUX;
    Se VALOR < INFO.AUX //procura posição na árvore
        Então AUX ← &AUX. ESQ; //valor < para a esquerda
        Senão AUX ← &AUX. DIR; // valor >= para a direita
    Fimse;
Fimenquanto;
ALOQUE (*AUX);
AUX.INFO ← VALOR; //insere valor
AUX.DIR ← null;
AUX.ESQ ← null;
Se INFO.AUX < INFO.ANT //liga novo nó na árvore
    Então ANT.ESQ ← *AUX;
    Senão ANT.DIR ← *AUX;
Fimse;
Fimse
FimInserir;

```

7.3.3 Buscar elemento em uma árvore

A busca de um elemento em uma árvore realizada de forma iterativa é similar à inclusão, já que na inclusão é necessário encontrar a posição correta na árvore binária de busca para inserir o novo elemento. Então o percurso é realizado a partir da raiz, sempre respeitando a regra da ordem de inserção: “maiores ou iguais para a direita” e “menores para a esquerda”.

```

Procedimento Buscar (RAIZ: NO, VALOR:inteiro)
Inicio
    AUX, PAI: NO;
    AUX ← *RAIZ;
    PAI ← null;
    enquanto AUX.INFO <> VALOR e AUX <> null faça
        PAI ← AUX;
        Se AUX.INFO < VALOR
            Então AUX ← *AUX.ESQ;
            Senão se AUX.INFO >VALOR
                Então AUX ←*AUX.DIR;

```

```
        Fimse;
    Fimse;
Fimenquanto;
Se AUX.INFO = VALOR
    Então imprima ("valor encontrado");
    Senão imprima ("valo não encontrado na estrutura");
Fimse;
FimBuscar;
```

7.3.4 Listar os elementos da árvore

Uma característica interessante nos algoritmos de manipulação de árvores é que algumas funções não podem ser realizadas de forma iterativa. A listagem de todos os elementos de uma árvore é um processo que só pode ser resolvido de forma recursiva. Isto ocorre porque não há decisão sobre para qual lado andar na árvore, se para o lado esquerdo ou direito. É preciso ir para os dois lados ao mesmo tempo, e isto só pode ser feito de forma recursiva.

7.3.5 Percorrer uma árvore binária de busca

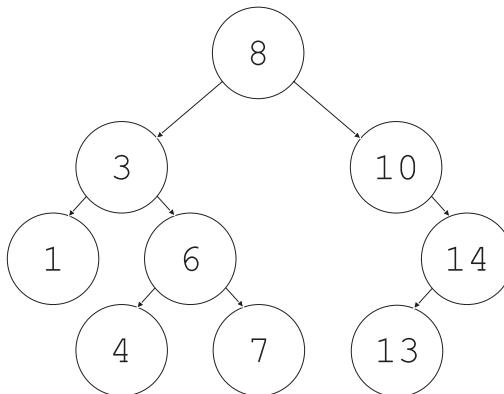
O percurso em árvores binárias de busca pode ser feito de três maneiras:

- a) pré-ordem – o percurso em pré-ordem determina que o conteúdo de um nó deve ser exibido antes que sejam exibidos os seus descendentes (nós filhos). O algoritmo visita o nó; percorre a subárvore da esquerda em pré-ordem; percorre a subárvore da direita em pré-ordem. Em resumo, a sequência é: raiz da subárvore – esquerda – direita.
- b) pós-ordem – o percurso em pós-ordem determina que o conteúdo de um nó somente seja exibido depois que forem exibidos os conteúdos de todos os seus descendentes (nós filhos). O algoritmo percorre a subárvore da esquerda em pós-ordem; percorre a subárvore da direita em pós-ordem; visita o nó. Em resumo, a sequência é: esquerda – direita – raiz da subárvore.
- c) em ordem – o percurso em ordem é o mais utilizado. Nesta forma, a regra é que sejam exibidos os descendentes da esquerda, para então ser exibido o conteúdo do próprio nó, para só então serem

exibidos os descendentes de sua direita. O algoritmo percorre a subárvore da esquerda em ordem, visita o nó; percorre a subárvore da direita em ordem; visita o nó. Em resumo, a sequência é: esquerda – raiz da subárvore – direita. Desta forma, os elementos da árvore são listados em ordem. Uma vez que os nós são inseridos na árvore respeitando a ordenação, o percurso e exibição em ordem torna-se natural.

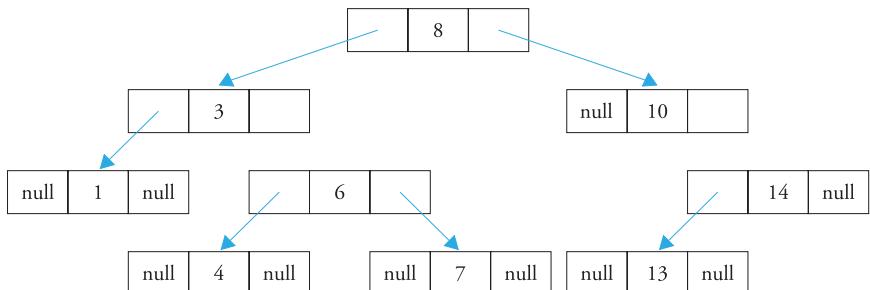
A figura 7.9 mostra uma árvore, que está representada na figura 7.10 na forma da estrutura de registros e ponteiros.

Figura 7.9 – Árvore para exemplificar os percursos



Fonte: Elaborada pelo autor.

Figura 7.10 – Árvore da figura 7.9 representada em forma de registros e ponteiros



Fonte: Elaborada pelo autor.

Estrutura de Dados

As três formas de percurso na árvore da figura 7.9 são:

- × resultado da sequência de visita aos nós da árvore da figura 7.9 no percurso pré-ordem: 8, 3, 1, 6, 4, 7, 10, 14, 13.
- × resultado da sequência de visita aos nós da árvore da figura 7.9 no percurso pós-ordem: 1, 4, 7, 6, 3, 13, 14, 10, 8.
- × resultado da sequência de visita aos nós da árvore da figura 7.9 no percurso em ordem: 1, 3, 4, 6, 7, 8, 10, 13, 14.

Exemplo de algoritmos recursivos para os percursos em árvores são apresentados na sequência.

```
Função PreOrdem (AUX: NO)
    Inicio
        Se AUX <> null
            Então imprima (AUX.INFO);
            PreOrdem (*AUX.ESQ);
            PreOrdem (*AUX.DIR);
        Fimse;
    FimPreOrdem;
```

```
Função PosOrdem (AUX: NO)
    Inicio
        Se AUX <> null
            Então PosOrdem (*AUX.ESQ);
            PosOrdem (*AUX.DIR);
            Imprima (AUX.INFO);
        Fimse;
    FimPosOrdem;
```

```
Função EmOrdem (AUX: NO)
    Inicio
        Se AUX <> null
            Então EmOrdem (*AUX.ESQ);
            Imprima (AUX.INFO);
            EmOrdem (*AUX.DIR);
        Fimse;
    FimEmOrdem;
```

7.3.6 Remover um elemento da árvore

Para remover um nó de uma árvore binária, o primeiro passo é localizar o nó que será removido e também o pai deste nó, ou seja, seu antecessor. Este algoritmo já conhecemos, e é o de busca de um elemento na árvore.

A partir daí, deve-se observar três situações diferentes quando o nó em questão é removido, pois cada um deles requer tratamento específico na remoção. As três situações possíveis são:

1. quando o nó a ser removido possuir filhos – neste caso, temos as outras duas situações de remoção:
 - 1.1 quando o nó a ser removido possuir apenas um nó descendente, ou seja, apenas um filho – basta apontar direto do pai do nó a ser removido para seu filho.
 - 1.2 quando o nó a ser removido tiver dois filhos – neste caso, não podemos aplicar a mesma regra do apontamento direto do pai do nó a ser removido para seu filho, pois agora são dois filhos. E também não podemos simplesmente utilizar o outro ponteiro do pai do nó a ser removido, pois assim estariamos comprometendo a ordem dos elementos. Então, para que a ordem seja garantida, deve-se substituir o nó a ser removido pelo filho que está mais à direita da subárvore da esquerda do nó a ser removido, realizando uma rotação.
2. quando o nó a ser removido é o último nó da árvore – nesta forma deve-se atualizar o ponteiro do nó raiz da árvore, tornando-o nulo.

```

Função Remove (RAIZ:NO, VALOR : inteiro) : inteiro
  Início
    ANT, ATUAL: NO;
    Se RAIZ = null
      Então retorne 0; //não existe a árvore
    Fimse;
    ANT ← null;
    ATUAL ← *RAIZ;
```

Estrutura de Dados

```
Enquanto ATUAL <> null faça
    Se VALOR = ATUAL.INFO //encontrou o nó a ser removido
        Então se ATUAL = *RAIZ //verifica o lado da remoção
            Então *RAIZ <- RemoveAtual (ATUAL); //remove a raiz
            Senão se ANT.DIR = ATUAL
                Então ANT.DIR ← RemoveAtual (ATUAL);
            //remove da direita
            Senão ANT.ESQ ← RemoveAtual (ATUAL);
            //remove da esquerda
            Fimse;
            Fimse;
            Retorne 1; //removeu o valor e finaliza a função
        Fimse;
        Fimenquanto;
        ANT ← ATUAL; //continua a buscar o valor a ser removido
        Se VALOR > ATUAL.INFO
            Então ATUAL ← *ATUAL.DIR;
            Senão ATUAL ← *ATUAL.ESQ;
        Fimse;
    FimRemove;

Função RemoveAtual (ATUAL:NO) : NO
    Inicio
        NO1, NO2: NO;
        Se ATUAL.ESQ = null //trata nó folha e nó com 1 filho
            Então NO2 ← ATUAL.DIR;
            Desaloque (ATUAL);
            Retorne (*NO2);
        Fimse;
        NO1 ← ATUAL; // procura filho mais à direita na subárvore esquerda
        NO2 ← ATUAL.ESQ;
        Enquanto NO2.DIR <> null
            NO1 ← NO2;
            NO2 ← NO2.DIR;
        Fimenquanto;
        Se NO1 <> ATUAL //copia o filho mais à direita na subárvore esquerda para
        o lugar do nó removido
            Entao NO1.DIR ← NO2.ESQ;
            NO2.ESQ ← ATUAL.ESQ;
        Fimse;
        NO2.DIR ← ATUAL.DIR;
        Desaloque (ATUAL);
        Retorne (*NO2);
    FimRemoveAtual;
```

Agora já temos todos os algoritmos das operações de manipulação de árvores binárias de busca e conseguimos utilizar esta estrutura para resolver diversas situações computacionais.

7.4 Aplicações práticas da estrutura de dados árvore

Há inúmeros problemas no mundo real que podem ser modelados e resolvidos por meio das árvores: estruturas de pastas de um sistema operacional, interfaces gráficas, bancos de dados e sites da internet, e até jogos digitais são exemplos de aplicações de árvores. O uso de estruturas de árvores ocorre com frequência quando a aplicação se baseia em percurso, na qual é necessário adotar uma ordem apropriada de percurso e utilizar uma rotina adequada de visita ao nó.

Entre as aplicações de árvores binárias, podem ser citadas as árvores de decisão usadas na inteligência artificial. Estruturas de dados do tipo árvore são bastante úteis em aplicações que necessitam tomada de decisão em determinado momento de um algoritmo.

Outra aplicação é na representação de expressões aritméticas. No caso da representação das expressões aritméticas, pode-se utilizar um caminhamento pós-fixado para resolver o problema, por exemplo.

Aplicações de mineração de dados, aplicações de pesquisa (busca), bancos de dados robustos armazenam os dados em formato de árvores, proporcionando maior velocidade de busca e agilizando o processamento. As árvores também podem ser utilizadas para resolver problemas de busca, como jogo da velha, damas e, além dos já citados, jogos digitais que temos disponíveis atualmente.

Saiba mais

Sobre o algoritmo de árvore de decisão para aplicações de mineração de dados, você encontra informações no link <<https://msdn.microsoft.com/pt-br/library/ms175312.aspx>>.

Outra árvore comumente encontrada é a estrutura hierárquica de arquivos em um sistema operacional. O diretório-raiz de um HD (referenciado como C:\, em muitos sistemas) é o nó-raiz da árvore. Os diretórios

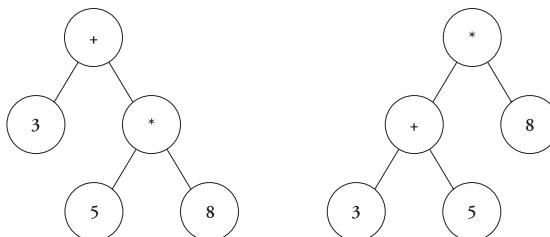
em um nível abaixo do diretório-raiz são seus nós-filhos. Pode haver muitos subdiretórios (imagens, vídeos, documentos, programas, entre outros). Os arquivos são os nós-folha da árvore, pois não têm filhos.

Síntese

Já são de nosso conhecimento algumas estruturas de dados, como pilha, fila e listas. Sabemos que, de acordo com o problema, escolher a estrutura de dados mais adequada costuma trazer benefícios quanto à economia de recursos computacionais. Em muitas aplicações, como em banco de dados, estruturas mais robustas são necessárias. Diante desta situação, surgem as árvores. Assim, neste capítulo, comentamos sobre a estrutura de dados do tipo árvore, uma das estruturas mais utilizadas nos algoritmos computacionais, pois agiliza o processamento de algoritmos complexos. Os diversos tipos de árvores apresentados têm função específica e obedecem a um padrão de comportamento. Foram apresentados os principais algoritmos de manipulação de nós na estrutura árvore considerando as árvores binárias de busca, que são as mais utilizadas nos recursos computacionais.

Atividades

1. Qual dos 3 percursos em árvore é o mais útil? Por quê?
2. As árvores parecem muito mais complicadas do que os vetores ou listas encadeadas. Elas são realmente úteis?
3. Precisamos reorganizar os nós sempre que inserimos um novo nó na árvore?
4. Considerando as duas árvores a seguir, escreva a expressão aritmética de cada uma delas com base no percurso em-ordem. Se preciso, utilizar () para definir a prioridade de execução.



8

Grafos

UMA ESTRUTURA DE dados não-linear bastante poderosa é o grafo. Assim como a árvore (ver capítulo 7), o grafo também é uma estrutura composta por nós, na qual cada nó aponta para outro nó. A diferença é que os nós de um grafo não ficam dispostos em uma estrutura hierárquica. Os nós são dispostos em uma malha em que cada nó pode apontar para zero, um ou mais nós do grafo, e também receber apontamento de qualquer outro nó, ou então nenhum apontamento.

ESTE TIPO DE estrutura é bastante utilizado quando rotas e mapas estão em constante uso pelos cidadãos do mundo. Por meio dos grafos é possível a implementação dos complexos sistemas de mapas e rotas em sistemas de orientação por satélite, os GPS.

Objetivos de aprendizagem:

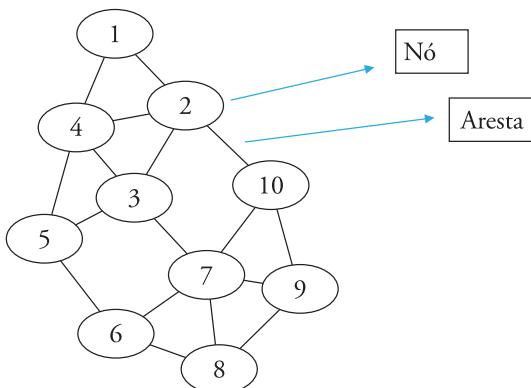
1. conhecer a estrutura de dados grafos e suas características;
2. aprender a representar um grafo nos algoritmos;
3. conhecer os algoritmos clássicos de manuseio de grafos.

8.1 Conceitos de grafos

O conceito de grafos vem de uma área da Matemática que se dedica a estudar as relações entre entidades (objetos) que possuem características relevantes. Com o desenvolvimento da área de Inteligência Artificial na Computação, a estrutura de dados grafos permite que se desenvolva algoritmos eficientes e eficazes nesta área. Para trabalharmos com grafos, precisamos conhecer as estruturas matrizes e listas, pois serão muito úteis na construção e manipulação de grafos pelos algoritmos.

Um grafo é composto por um conjunto de nós e arestas. Um nó, vértice ou ponto representa uma entidade no grafo, que pode ser por exemplo, uma fruta, uma cidade, uma pessoa. Uma aresta, arco ou linha é uma relação que liga dois nós, que pode ser uma estrada ligando cidades, ou um grau de parentesco ligando pessoas, por exemplo. Uma aresta é representada por um par ordenado de nós. A figura 8.1 mostra um grafo com seus nós e arestas, por exemplo o nó 2, o nó 10 e a aresta (2,10) que une os nós 2 e 10.

Figura 8.1 – Um grafo com seus nós e arestas



Fonte: AdalbertoFelipe.wordpress.com.

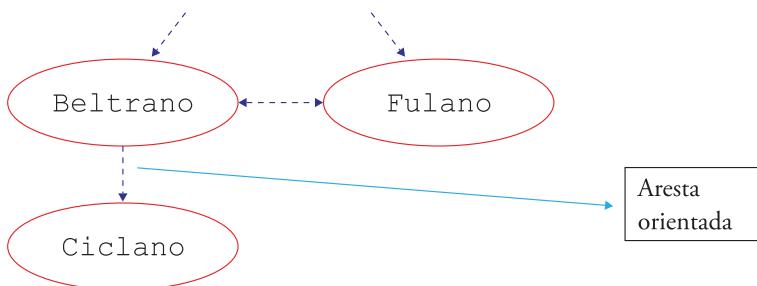
Para designar um grafo, utilizamos uma notação conforme segue:

$$G(n, a)$$

Na qual G é o nome do grafo, n é o número de nós e a é o número de arestas deste grafo. No exemplo da figura 8.1 temos um grafo G com 10 nós (1 até 10) e 17 arestas ligando estes nós. Logo, podemos representar este grafo como $G(10, 17)$.

A aresta de um grafo pode ser orientada. Isto ocorre quando, para representar a função que forma o par ordenado do grafo (relação entre os nós), existe uma sequência ou precedência. Por exemplo, no grafo que representa o organograma de uma empresa, podemos ter que Beltrano é chefe de Ciclano. Beltrano e Ciclano são os nós do grafo (organograma), e a aresta orientada representa a ordem hierárquica. Os grafos que apresentam arestas orientadas são chamados de grafos orientados ou dígrafos. A figura 8.2 mostra um organograma de uma empresa em que as arestas orientadas representam a ordem hierárquica entre os funcionários.

Figura 8.2 – Exemplo de um grafo orientado ou dígrafo



Fonte: <http://marciofibonacci.blogspot.com.br/>.

O grau de um nó é o número de arestas ligadas a este nó. No exemplo da figura 8.1 temos que o nó 7 tem grau 5, enquanto que o nó 1 tem grau 2. No exemplo da figura 8.2, podemos perceber que o nó Chefe tem grau 2 e o nó Ciclano tem grau 1.

Porém, os grafos orientados permitem mais características em relação ao grau de seus nós:

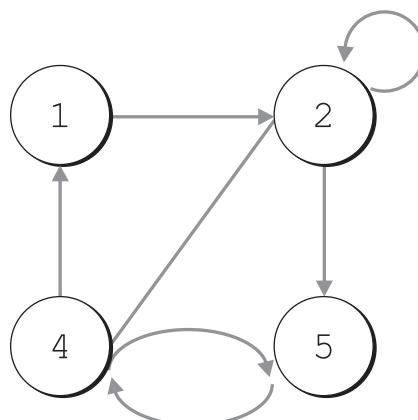
Estrutura de Dados

- × grau de entrada (ou de recepção) de um nó, é o número de arestas que tem este nó como destino. Os nós com grau de entrada 0 são chamados de fonte;
- × grau de saída (ou de emissão) de um nó, é o número de arestas que tem este nó como origem. Os nós com grau de saída 0 são chamados de sumidouros;
- × um laço, é uma aresta que une o nó a si próprio, isto é, o vértice aponta para ele mesmo;
- × em um grafo regular, os nós têm todos o mesmo grau;
- × em um grafo completo, existe uma aresta entre cada par de nós do grafo, isto é, todos os vértices estão ligados.

No exemplo da figura 8.2, que representa um grafo orientado, temos que o nó Chefe tem grau de entrada 0, logo é um nó Fonte; e o nó Ciclano tem grau de saída 0, então é um nó Sumidouro.

Temos outros conceitos em um grafo, como por exemplo sua ordem. A ordem de um grafo é dada pela cardinalidade do conjunto de vértices do grafo, isto é, o total de vértices (nós) que o grafo tem. Portanto, o grafo da figura 8.1 tem ordem 10, e o grafo da figura 8.2 tem ordem 4.

Figura 8.3 – Representação de laço, ciclo e caminho em um grafo orientado



Fonte: Montebello Junior (2014).

Pode haver caminhos em um grafo orientado. O *caminho* é uma sequência de nós interligados, ligando um nó (origem) a um outro nó (destino). Podemos dizer que um caminho em um grafo orientado é formado por X nós e $X-1$ arestas, e que o comprimento de um caminho equivale a seu número de arestas. A figura 8.3 mostra um grafo orientado com 4 arestas e diversos caminhos, como por exemplo: (1, 2, 4, 1); (1, 2, 5, 4, 5); (1, 2, 5, 4, 1); (1, 2, 2, 4, 1) e os outros.

É possível classificar um caminho como simples quando todos os nós deste caminho são distintos, isto é, o caminho simples não passa por um nó mais de uma vez. Outro conceito importante de caminho é o ciclo. O *ciclo* é um caminho cuja origem é igual ao destino e o comprimento é maior ou igual a 2. Já um *laço* é um caminho com origem e destino iguais e comprimento 1. A figura 8.3 mostra um laço no nó 2, com comprimento igual a 1; e um ciclo no caminho formado pelos nós 1, 2, 4, com comprimento maior do que 2, no caso comprimento igual a 3.

Mais alguns conceitos em relação aos grafos:

- ✗ um grafo pode ser acíclico quando não possuir nenhum ciclo e, consequentemente, um grafo cíclico possui pelo menos um ciclo.
- ✗ um grafo é considerado conexo quando possuir pelo menos um nó de onde partem todos os outros nós, que basicamente forma uma árvore (ver capítulo 7).
- ✗ um grafo é fortemente conexo quando todos os nós têm a propriedade anterior.

Podemos representar o mapa do metrô de uma cidade por meio de um grafo, e daí traçar rotas entre as estações, considerando o tempo, a distância, o trânsito (de pessoas neste caso) etc. A figura 8.4 mostra o mapa do metrô de São Paulo, no qual cada estação pode ser um nó do grafo e cada linha de metrô entre duas estações pode ser a aresta ligando os nós, neste caso, como o metrô vai e volta nos trilhos, o grafo é não-orientado.

Estrutura de Dados

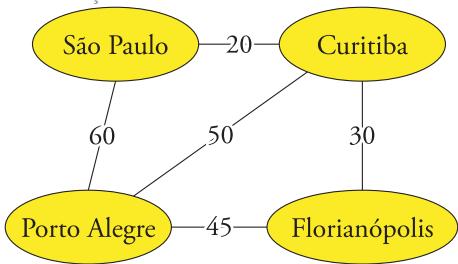
Figura 8.4 – Mapa do metrô da cidade de São Paulo



Fonte: www.metro.sp.gov.br.

Para podermos identificar o tempo ou a distância entre as estações de uma linha do metrô, precisamos guardar esta informação na aresta que liga os nós do grafo (linha do metrô que liga as estações). Neste caso, teremos um grafo valorado. Veja o exemplo de um grafo valorado na figura 8.5, na qual os valores nas arestas podem representar a distância entre as cidades que constam nos nós.

Figura 8.5 – Grafo valorado mostrando informações entre as cidades nas arestas



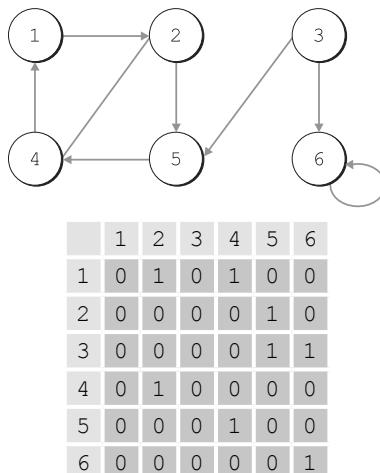
Fonte: <http://www.inf.ufsc.br/grafos/definicoes/definicao.html>.

8.2 Representação de um grafo

A partir dos conceitos citados, podemos classificar, organizar e utilizar grafos para resolver problemas em algoritmos. Mas para tanto, é necessário representar esta estrutura nos algoritmos. É possível representar a estrutura de um grafo utilizando matriz de adjacência (ver capítulo 1) ou listas encadeadas (ver capítulo 5).

Uma matriz $M \times N$ é uma matriz de adjacência quando o elemento $M[i,j]$ é igual a 1 para indicar que existe uma aresta ligando o nó i ao nó j , e igual a 0 quando não existe tal aresta no grafo que a matriz de adjacência representa. A figura 8.6 mostra um exemplo de um grafo representado na forma de matriz de adjacência. Os 6 nós estão representados nos índices de linha e coluna e as arestas nos valores 1 da matriz, em que os índices representando os nós se cruzam. Por exemplo, o nó 1 tem valor 1 nas posições [1,2] e [1,4] da matriz, mostrando as arestas entre estes nós, e zero nas demais posições, em que não existe ligação no grafo orientado ou direcionado.

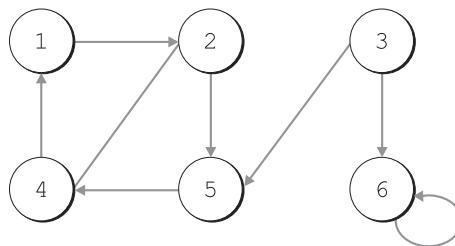
Figura 8.6 – Grafo direcionado representado em forma de uma matriz de adjacência



Fonte: Montebello Junior (2014).

Para representar um grafo na forma de uma lista encadeada, utiliza-se uma combinação de listas. A lista principal possui uma lista de nós do grafo, e cada nó está ligado a uma lista de vértices (nós), com os quais o nó forma uma aresta. A figura 8.7 mostra um exemplo de um grafo direcionado representado na forma de uma lista encadeada, em que a primeira lista, dita principal, contém os 6 nós do grafo, e a cada nó da lista estão ligadas listas que contêm os nós com os quais o nó da lista principal forma uma aresta. Por exemplo, o nó 1 faz aresta com os nós 2 e 4, obedecendo a orientação do grafo.

Figura 8.7 – Grafo direcionado representado em forma de lista de adjacência



1	[]	\rightarrow	[]	\rightarrow	[4 /]
2	[]	\rightarrow	[5 /]	\rightarrow	
3	[]	\rightarrow	[6 /]	\rightarrow	[5 /]
4	[]	\rightarrow	[2 /]	\rightarrow	
5	[]	\rightarrow	[4 /]	\rightarrow	
6	[]	\rightarrow	[6 /]	\rightarrow	

Fonte: Montebello Junior (2014).

Quando temos um grafo valorado como o da figura 8.5, para representarmos estes valores na matriz de adjacência substituímos o valor 1 da existência de uma aresta entre os nós pelo valor da aresta, no caso da figura 8.5 pela distância entre as cidades. Esta representação na lista de adjacência pode ser feita criando um campo no registro das sublistas, que identifica as arestas na lista, com o valor da aresta. Neste caso, o nó da sublista teria os campos do valor do nó, do valor da aresta e do ponteiro para o próximo nó da lista encadeada.

8.3 Algoritmos para representação de grafo

Agora vamos conhecer os algoritmos que permitem que os grafos sejam armazenados nas estruturas de matriz de adjacência e na lista de adjacência.

8.3.1 Matriz de adjacência

Uma matriz de adjacência é uma forma de dispor os vértices do grafo em uma matriz quadrada, onde linhas e colunas são compostas pelos vértices (nós) do grafo. Uma das dimensões da matriz irá representar os vértices (nós) de origem, enquanto a outra dimensão irá representar os vértices (nós) de destino. A interseção entre linha e coluna representa a adjacência. Desta maneira, sempre que se quiser saber o destino de um determinado vértice, basta olhar a linha correspondente na matriz. Ao mesmo tempo, a matriz evidencia quais são as origens de um vértice. Basta dar uma olhada na coluna correspondente ao vértice em questão e esta informação poderá ser facilmente constatada. A matriz de adjacências é um recurso que apresenta de uma forma geral as adjacências do grafo como um todo.

Para representar que existe uma aresta entre os vértices (linha e coluna), colocamos o valor 1 no conteúdo da matriz na interseção dos vértices (linha e coluna). Caso não exista uma aresta entre os vértices, o valor no conteúdo da matriz é 0.

Veja a matriz na figura 8.6 e na sequência o algoritmo para montá-la. Perceba que, como o grafo da figura 8.6 é orientado, então a matriz de adjacência terá a aresta a partir do nó origem para o nó destino. Isto é, a aresta (1,2) será representada na matriz como grafo [1,2] = 1, e grafo [2,1] = 0. Caso o grafo não seja orientado, teremos grafo [1,2]=1 e grafo [2,1]=1.

```

Algoritmo // matriz de adjacência
inicio
    TIPO M = MATRIZ [1:6,1:6] inteiro;
    GRAFO : M;
    I, J, ORIGEM, DESTINO :inteiro;
    RESP: caractere [1];
    Para I de 1 até 6 faça
        Para J de 1 até 6 faça
            GRAFO[I,J] ← 0; //não tem aresta
        Fimpara;
    Fimpara;
```

Estrutura de Dados

```
Fimpara;  
RESP ← "S";  
Enquanto RESP = "S" faça  
    Imprima("digite o nó origem da aresta");  
    Leia (ORIGEM);  
    Imprima("digite o nó destino da aresta");  
    Leia (DESTINO);  
    GRAFO[ORIGEM,DESTINO] ← 1; //insere uma aresta  
    Imprima ("existem mais arestas no grafo? S/N ")  
    Leia (RESP);  
Fimenquanto;  
Fim.
```

8.3.2 Lista de adjacência

A lista de adjacência é uma informação pertencente a cada um dos vértices de origem. A lista de adjacências de um vértice apresenta o conjunto de vértices com os quais este vértice se conecta, respeitando a direção quando o grafo for direcionado. Então, cada vértice de origem possuirá uma lista de destinos. Isto significa que não podemos ter um único campo na estrutura do vértice para armazenar este dado. E o próprio nome “lista de adjacências” já sugere que cada vértice terá uma sublista encadeada própria para armazenar as adjacências. Cada registro desta sublista irá comportar o endereço do vértice com o qual o vértice de origem estabelece conexão, ou seja, o seu vértice destino.

Cada registro na lista possui pelo menos dois campos:

1. informação do vértice – que armazena o índice (valor) do vértice que é adjacente;
2. próximo vértice – que é um ponteiro para o próximo nó adjacente.

As cabeças das listas podem ser armazenadas em um vetor de ponteiros para facilitar o acesso aos vértices.

Os registros declarados para a cabeça da lista e dos nós da lista podem vir a conter mais campos, conforme a necessidade do algoritmo, como por exemplo um campo flag indicando se o nó já foi visitado (1) ou não (0), um campo indicando o valor da aresta (como no grafo da figura 8.5), entre outras necessidades.

Visitar um nó significa que o nó é marcado, de modo que, se for encontrado novamente, no caso de um percurso, não será visitado outra vez. Esta situação também é útil quando se utiliza a informação contida no nó, não fazendo uso da informação em duplicidade, quando for o caso.

Veja o grafo e as listas de adjacência na figura 8.7 e, na sequência, o algoritmo para implementá-lo utilizando listas encadeadas.

Representação das estruturas de cabeçalho e da lista de adjacência:

```

Tipo NO = registro
    INFO: inteiro; //pode haver mais campos se necessário
    *PROX: ponteiro para NO;
FimNO;
REG: NO // cabeçalho da lista de adjacência

Algoritmo // lista de adjacência
inicio
    Tipo NO = registro
        INFO: inteiro;
        *PROX: ponteiro para NO;
    FimNO;
    REG: NO;
    ATUAL, NOVO: NO;
    RESP: caractere [1];
    RESP ← "S";
    Aloque (&REG); //iniciando o cabeçalho da lista
    Enquanto RESP = "S" faça
        Imprima("digite o nó origem da aresta");
        Leia (ORIGEM);
        Se *REG <> nulo //busca origem no cabeçalho
            Então ATUAL ← *REG;
            Enquanto ATUAL.*PROX <> nulo E ATUAL.INFO
            <> ORIGEM faça
                ATUAL ←*(ATUAL.*PROX);
            FimEnquanto;

            Se ATUAL.*PROX = nulo //insere novo nó no cabeçalho
                Então aloque (&NOVO);
                ATUAL.*PROX <- &NOVO;
                NOVO.INFO ← ORIGEM;
                NOVO.*PROX ← nulo;
                ATUAL ← NOVO;

```

```
Fimse;
    //ATUAL contém o vértice origem no cabeçalho
Imprima("digite o nó destino da aresta");
Leia (DESTINO);
Se ATUAL.*PROX <> nulo //insere destino na lista de adjacência a
partir do cabeçalho
Então //encontra final da lista de adjacência
    Enquanto ATUAL.*PROX <> nulo
        ATUAL ←*(ATUAL.*PROX);
    Fimenquanto;
    aloque (&NOVO);
    ATUAL.*PROX ← &NOVO;
    NOVO.INFO ← DESTINO;
    NOVO.*PROX ← nulo;
Senão //primeiro nó da lista de adjacência de ORIGEM
    Aloque (&NOVO);
    ATUAL.*PROX ← &NOVO;
    NOVO.INFO ← DESTINO;
    NOVO.*PROX ← nulo;
Fimse;
imprima ("existem mais arestas no grafo? S/N ")
Leia (RESP);
Fimenquanto;
Fim.
```

Em função da possibilidade de utilizarmos alocação dinâmica nas listas de adjacência, esta forma de representação de grafos torna-se mais interessante, pois permite que seja aumentada ou diminuída sua estrutura com facilidade. Por este motivo, iremos utilizar esta forma de representação nos próximos algoritmos de grafos.

8.4 Caminhos em grafos

A partir do momento em que temos o grafo mapeado em uma lista de adjacência, podemos identificar caminhos em nosso grafo, e é aqui que os grafos têm a importância para os algoritmos de mapas e localizações, por exemplo. Para percorrermos um grafo mapeado em uma lista de adjacência, basta seguir o encadeamento da lista gerada pelo grafo a partir dos ponteiros dos nós.

Podemos ter um campo no registro do nó com o peso da aresta. Este peso pode ser a distância ou o tempo de percurso entre duas cidades (repre-

sentadas pelos nós ligados pela aresta), o custo para irmos de um ponto ao outro (identificado pelos nós) etc. Estes pontos podem representar etapas de um projeto, filiais de uma empresa, entre outras informações. Normalmente, temos interesse em encontrar a menor distância, o menor tempo, o menor custo, e para isto os diversos algoritmos de caminho em grafos podem ser utilizados.

O percorrer um grafo em algoritmos significa visitar todos os vértices de um grafo. O percurso em árvores é um caso particular de percurso em grafo. Da mesma forma que em árvores, o percurso em um grafo pode ser realizado em profundidade e em largura, também conhecido como busca em profundidade e busca em largura.

A busca em profundidade, num contexto de árvore, visita os nós-filhos antes de visitar os nós-irmãos. No contexto de grafos, implementa-se uma pilha e visita-se os vértices adjacentes em uma ordem particular.

Já a busca em largura, num contexto de árvores, visita os nós-irmãos antes de visitar os nós-filhos. No contexto de grafos, implementa-se uma fila e permite-se, ao visitar o vértice (retirando da fila), armazenar em cada vértice qual sua distância do vértice inicial.

Para a sequência de passos dos algoritmos de percurso, ambos os casos partem de um vértice escolhido arbitrariamente e visita-se este vértice. Em seguida, considera-se cada um dos nós W adjacentes a V.

8.4.1 Percurso em profundidade

No percurso em profundidade, logo que é encontrado o primeiro W adjacente a V:

- visita-se o nó W fazendo uso de sua informação e marcando como visitado;
- coloca-se o nó V em uma pilha;
- faz-se V receber o conteúdo de W ($V <- W$).

A consequência do passo c (acima) é que o processo de considerar nós adjacentes ao antigo nó V é interrompido, passando-se a procurar nós adjacentes a W. Com a repetição desse processo vai-se o mais “longe” possível no

grafo (percurso em profundidade). A marcação de nós visitados pode ser realizada por meio de um campo flag no registro (não visitado = 0 e visitado = 1).

Após visitar um nó e ele será o novo nó V, se ele não tiver nós adjacentes ainda não visitados, o novo nó V será aquele colocado no topo da pilha, e recomeça o processo de considerar os nós adjacentes a W. Isto significa subir um nível no grafo.

Na sequência é apresentado um esboço do algoritmo de percurso em profundidade, que utiliza funções já descritas neste material.

Procedimento profundidade (V:nó)

```
T, W: nó;  
P:pilha;  
visite (V); //utiliza a informação do nó  
marque (V); //torna campo flag=1  
empilhe (V,P);  
Enquanto P não for vazia faça  
    T ← topo (P);  
    desempilhe (P);  
    para cada W adjacente a T faça  
        se W não for marcado  
            então visite (W);  
            marque (W);  
            empilhe (T,P);  
            T ← W;  
        fimse;  
    fimpara;  
fimenquanto;  
fimProfundidade.
```

8.4.2 Percurso em largura

No percurso em largura, para cada um dos nós W:

- visita-se o nó W;
- coloca-se o nó W em uma fila.

Ao terminar de visitar os nós W, toma-se o nó que estiver na frente da fila e repete-se o processo visitando cada um dos nós adjacentes a ele, colocando-os na fila. A visita ao nó ocorre conforme comentado no item 8.4.1.

Segue o esboço do algoritmo de percurso em largura, que utiliza funções já descritas neste material.

```

Procedimento largura (V:nó)
    T,W: nó;
    F:fila;
    visite (V);
    marque (V);
    enfileire (V,F);
    enquanto F não for vazia faça
        T ← frente (F);
        desenfileire (F);
        para cada W adjacente a T faça
            se W não for marcado
                então      visite (W);
                marque (W);
                enfileire(W,F);
            fimse;
        fimpara;
    fimenquanto;
fimlargura;
```

É interessante observar semelhança entre os dois procedimentos de busca. Ambos apresentam duas repetições: a mais externa, que repete até a estrutura auxiliar (pilha ou fila) estar vazia; e a mais interna, que pega sucessivamente os nós adjacentes ao nó T que está sendo considerado. No caso do percurso em profundidade, a repetição mais interna é interrompida.

Síntese

Neste capítulo vimos a estrutura de dados grafo. Esta forma de representação é bastante utilizada em nosso cotidiano, mas para utilizá-la de forma computacional é preciso armazená-la utilizando estruturas de dados mais simples, como matriz de adjacência e lista de adjacência. A partir desta representação é que conseguimos utilizar esta estrutura nos algoritmos.

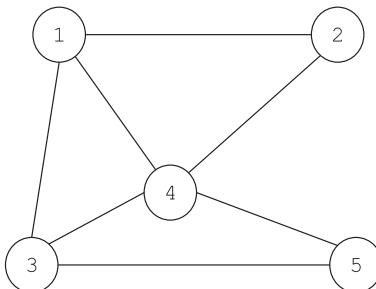
Diversos algoritmos para cálculos complexos fazem uso de grafos, principalmente utilizando os direcionados e valorados, para obterem informações relevantes.

Atividades

1. Desenhe um grafo com no mínimo 6 vértices, orientado. Mostre exemplos de:
 - a) 2 nós;
 - b) 2 arestas;
 - c) 2 percursos;
 - d) 1 ciclo;
 - e) 1 laço.
2. Escreva um algoritmo que imprima o grau de cada vértice de um grafo em uma matriz de adjacência.
3. A partir da matriz de adjacência a seguir, desenhe o grafo:

	1	2	3	4	5	6
1	0	0	1	1	1	0
2	0	0	1	1	1	0
3	1	1	0	0	1	0
4	1	1	0	0	1	0
5	1	1	1	1	0	0
6	0	0	0	0	0	0

4. Mostre a lista de adjacência para o grafo a seguir:



9

Métodos de Ordenação e Pesquisa

ATUALMENTE DISPOMOS DE uma grande quantidade de informação armazenada em bancos de dados. Estas informações sozinhas não têm grande significado, porém se adequadamente organizadas podem ser de grande valia para pesquisas e entendimento do que acontece ao nosso redor.

MAS PARA QUE estas informações sejam organizadas, é preciso que se tenha algoritmos que realizem esta tarefa. Pesquisadores renomados criaram algoritmos e continuam buscando soluções eficientes para organizar e buscar informações em bases de dados. Estes algoritmos estão disponíveis para aprendermos e utilizarmos em nosso software.

TAMBÉM EXISTE GRANDE demanda de aplicações que envolvem a pesquisa e busca por determinada informação, como as pesquisas realizadas pelas máquinas de busca disponíveis na internet. Neste capítulo iremos conhecer estas ferramentas e suas aplicações. Para entendermos estes algoritmos será utilizada a estrutura de dados Vetor para armazenar os dados a serem classificados e pesquisados.

Objetivos de aprendizagem:

1. conhecer os algoritmos clássicos de ordenação e classificação de vetores;
2. conhecer os algoritmos clássicos de pesquisa em vetores.

9.1 Classificação de vetores

A classificação ou ordenação aborda as técnicas utilizadas para classificar o conteúdo de um vetor em ordem crescente ou decrescente, numérica ou alfabética. Existem diversos algoritmos prontos e comprovados para realizar tal tarefa. Neste capítulo serão apresentados três destes algoritmos: método bolha, seleção simples e inserção.

Os métodos de ordenação apresentados podem ser utilizados para ordenar vetores numéricos ou alfabéticos. Uma vez que cada caractere possui um valor diferente do outro, a letra “A” tem valor menor do que a letra “B”, e assim por diante. Se a letra “A” for comparada com a letra “a”, ambas também terão valores diferentes. Cada caractere (numérico, alfabético ou caractere especial) é guardado dentro da memória de um computador segundo o valor de um código. Este código recebe o nome de ASCII (*American Standard Code for Information Interchange* – Código Americano Padrão para Troca de Informações). E é com base nesta tabela que o processo de ordenação trabalha, pois cada caractere tem um peso, um valor previamente determinado, segundo este padrão (KANTEK; DE BASSI, 2013).

Na sequência serão apresentados algoritmos clássicos de ordenação de vetores.

9.1.1 Método bolha

O método bolha é um dos mais clássicos métodos de ordenação conhecidos. Este método utiliza uma lógica bastante simples, porém seu processamento é lento. A seguir será apresentada a semântica do método e seu algoritmo.

× Semântica do método bolha

O algoritmo do método bolha aplica a seguinte lógica: uma vez comparados dois valores adjacentes no vetor, se eles estiverem na

ordem desejada, são mantidos em seus lugares e, se não estiverem, são invertidos. A execução sucessiva deste processo faz com que os menores valores comecem a subir aos poucos (ordem ascendente) e consequentemente os maiores comecem a descer, daí a analogia com uma bolha de sabão – método bolha.

✖ Sintaxe do método bolha

O algoritmo do método bolha apresentado aqui utiliza uma estrutura vetor de 5 posições e ordena o conteúdo do vetor em ordem crescente, isto é, do menor para o maior valor. Os comentários inseridos no algoritmo visam seu melhor entendimento, explicando o que está sendo realizado naquele trecho.

```

inicio //método bolha
    tipo VET = VETOR [ 1:5 ] inteiro;
    V: VET;
    I, AUX, LIM : inteiro;
    LIM ← 5; //tamanho do vetor
    leia (V);
    enquanto LIM ≥ 2 faça
        I ← 2; //posiciona no segundo elemento do vetor
        enquanto I ≤ LIM faça
            se V[I] < V[I-1] //compara o elemento anterior com o elemento posterior
                do vetor
                    então AUX ← V[I-1]; //troca os elementos
                    V[I-1] ← V [I];
                    V[I] ← AUX;
                fimse;
                I ← I +1;
            fimenquanto;
            LIM ← LIM -1;
        fimenquanto;
        imprima (V);
    fim.

```

Fonte: Kantek; De Bassi (2013).

Para que este algoritmo realize a classificação em ordem decrescente de seu conteúdo, é preciso inverter o sinal na comparação entre os valores adjacentes. Isto é, se o valor da posição I for maior do que o valor da posição I

– 1, então estes valores precisam ter suas posições trocadas – se $V[I] > V[I-1]$.

9.1.2 Método de ordenação oscilante

O método de ordenação oscilante é uma versão do método bolha, porém modificado e melhorado. Vamos verificar como ele funciona.

- ✗ **Semântica do método de ordenação oscilante**

A ideia de comparação entre valores e troca entre posições adjacentes se mantém, porém em vez de sempre ler o vetor na mesma direção, podemos inverter a direção entre passos subsequentes. Desta forma, elementos muito fora de suas posições irão mais rapidamente para suas posições corretas.

- ✗ **Sintaxe do método de ordenação oscilante**

Este método faz uso de *flag* (bandeira, em inglês), que sinaliza alguma situação no algoritmo, para controlar se houve ou não a troca entre valores no vetor. O algoritmo apresentado aqui utiliza um vetor de 5 posições e ordena o conteúdo em ordem crescente.

```
inicio // método ordenação oscilante
    tipo VET = VETOR [ 1:5 ] inteiro;
    V: VET;
    I, AUX, LIM, TROCA : inteiro;
    LIM ← 5; //tamanho do vetor
    leia (V);
    TROCA ← 1; //flag determinando que houve troca
    Enquanto TROCA = 1 e LIM >= 2 faça
        TROCA ← 0; //flag determinando que NÃO houve troca
        para I de LIM até 2 passo - 1 faça
            se (V[I - 1] > V[I])
                então
                    AUX ← V[I - 1];
                    V[I - 1] ← V[I];
                    V[I] ← AUX;
                    TROCA ← 1; //flag determinando que houve troca
                Fimse;
            fimpara;
            se TROCA = 1 //se houve troca
```

```

    então
TROCA ← 0; //flag determinando que NÃO houve troca
    para I de 2 até LIM faça
        se (V[I - 1] > V[I])
            então
                AUX ← V[I -1];
                V[I - 1] ← V[I];
                V[I] ← AUX;
                TROCA ← 1; //flag determinando que houve troca
            fimse;
        fimpara;
fimse;
LIM ← LIM -1;
fimenquanto;
imprima (V);
fim.

```

Novamente, para que este algoritmo realize a classificação em ordem decrescente de seu conteúdo, é preciso inverter o sinal na comparação entre os valores adjacentes. Isto é, se o valor da posição $I - 1$ for menor do que o valor da posição I , então estes valores precisam ter suas posições trocadas – se $(V[I - 1] < V[I])$ – nas duas partes do algoritmo.

9.1.3 Método seleção simples

O próximo método que veremos é o de seleção simples. Este também é um método clássico de ordenação e utiliza uma lógica interessante que agiliza o processamento da ordenação. A seguir será apresentada a semântica do método e seu algoritmo.

- × **Semântica do método de seleção simples**

A lógica deste método aplica uma pesquisa no vetor buscando qual o maior (ou menor) valor e selecionando-o. Então deve-se colocá-lo no fim (ou início) do vetor, já posicionando o elemento em seu local em relação a ordem desejada. E daí refaz-se o processo desconsiderando o elemento que foi alterado por último.

- × **Sintaxe do método de seleção simples**

O algoritmo do método de seleção simples apresentado utiliza a estrutura vetor de cinco posições e ordena o conteúdo do vetor

em ordem crescente, isto é, do menor para o maior valor. Desta forma, o algoritmo seleciona o menor valor do vetor e posiciona na primeira posição, e assim sucessivamente, até que todos os valores estejam corretamente posicionados conforme a ordem desejada.

As variáveis *indcor* e *corr* guardam os valores referentes ao índice do valor corrente e o valor corrente, respectivamente, auxiliando na guarda e indicação do menor valor e depois na troca de conteúdo do vetor. Verifique atentamente os comentários inseridos no algoritmo que visam um melhor entendimento, explicando o que está sendo realizado naquele trecho.

```
inicio // seleção simples
    tipo VET = Vetor [1:5] inteiro;
    V:VET;
    LIM, INDCOR, I, J: inteiro;
    CORR: inteiro;
    LIM ← 5; //tamanho do vetor
    I ← 1; //posiciona no primeiro elemento do vetor
    enquanto I < LIM faça
        CORR ← V[I];
        INDCOR ← I;
        J ← I + 1;
        enquanto J ≤ LIM faça //pesquisa o menor valor
            se V[J] < CORR
                então CORR ← V[J];
                INDCOR ← J;
            fimse;
            J ← J + 1;
        fimenquanto;
        AUX ← V[I]; //troca o menor valor pela 1ª. posição do vetor
        V[I] ← V[INDCOR];
        V[INDCOR] ← AUX;
        I ← I + 1; //posiciona para comparar a próxima posição
    fimenquanto;
fim.
```

Para que este algoritmo realize a classificação em ordem decrescente de seu conteúdo, é preciso buscar o maior valor do vetor e colocar este valor na primeira posição do vetor – se $V[J] > CORR$ – e assim sucessivamente a partir do último valor posicionado.

9.1.4 Método de inserção

O método de inserção tem uma proposta diferente, ele utiliza uma posição do vetor como variável auxiliar para ordenar o conteúdo do vetor. A seguir será apresentada a semântica do método e seu algoritmo.

- ✖ **Semântica do método de inserção**

Na lógica utilizada, o ciclo inicia no segundo elemento. O elemento é retirado do conjunto de dados, fazendo uso de uma posição auxiliar do vetor, e inserido em seu lugar definitivo, por meio da pesquisa de sua localização correta e do deslocamento dos outros elementos do conjunto.

- ✖ **Sintaxe do método de inserção**

A proposta deste algoritmo faz uso da posição 0 do vetor, desta forma o vetor de cinco posições que será ordenado precisa ser definido com seis posições (de 0 até 5). Por meio da logica entre os índices do vetor é encontrada a posição em que o elemento retirado dever ser inserido, desloca-se os demais elementos e insere-se o valor retirado em sua posição correta, e assim sucessivamente até que todos os valores tenham sido posicionados de acordo com a ordem desejada.

```

inicio //método de inserção
    tipo VET = Vetor [0:5] inteiro; //posição 0 do vetor é a posição auxiliar
    V:VET;
    LIM, I, J: inteiro;
    LIM ← 5; //tamanho do vetor
    para I de 2 até LIM faça
        V[0] ← V[I]; //retira o elemento do conjunto de dados
        J ← I - 1;
        enquanto V[J] < V[0] faça //pesquisa a localização correta deslocando os
        outros elementos
            V[J+1] ← V[J];
            J ← J - 1;
        fimenquanto;
        V[J + 1] ← V[0]; //insere o elemento retirado na posição correta
    fimpara;
fim.

```

Para que este algoritmo realize a classificação em ordem decrescente de seu conteúdo, é preciso encontrar a posição de inserção do valor retirado invertendo-se o sinal da pesquisa – enquanto $V[J] > V[0]$ faça – e assim sucessivamente para cada valor retirado do vetor.

9.1.5 Método Quick Sort

O Quick Sort é um dos métodos mais rápidos de ordenação de vetores, porém se as partes envolvidas na ordenação estiverem desequilibradas, isto pode gerar um processamento mais lento. A técnica utilizada neste método é a máxima de programação “dividir para conquistar”. Esta máxima diz que a divisão de um problema inicial grande (ou difícil) em subproblemas menores (ou mais simples) facilita para se atingir o objetivo final. Dada a lógica do método, é possível utilizar um algoritmo recursivo.

- × **Semântica do método Quick Sort**

O método Quick Sort baseia-se na divisão do vetor em dois subvetores, dependendo de um elemento chamado pivô, que normalmente será o primeiro elemento do vetor. O primeiro subvetor contém os elementos menores que o pivô, e o segundo subvetor contém os elementos maiores que o pivô. O pivô é colocado entre os dois vetores, já na posição correta em relação à ordenação desejada. Os dois subvetores são ordenados da mesma forma até que se chegue em um só elemento.

- × **Sintaxe do método Quick Sort**

Na sequência, iremos identificar as condições de parada e de recursão para então apresentar o algoritmo recursivo, que é o mais utilizado.

A condição de parada ocorrerá:

- × se o número de elementos a ordenar for 0 ou 1 então terminar.

A condição de recursão será:

- × ordenar os subvetores esquerdo e direito, usando o mesmo método recursivamente;

- × selecionar o elemento pivô “P”, escolhendo um elemento qualquer entre os elementos a ordenar;
- × processo de divisão: dividir os elementos em dois subvetores disjuntos, onde no subvetor 1 estarão os elementos inferiores ao pivô e no subvetor 2 estarão os elementos superiores ao pivô.

```

Função QuickSort (X[], IniVet, FimVet)
    X[ ] : vetor de inteiro;
    IniVet, FimVet: inteiro;
    Inicio
        I, J, PIVO, AUX: inteiro;
        I ← IniVet;
        J ← FimVet;
        Pivô ← X [(IniVet + FimVet) div 2]
        Enquanto I < J faça
            Enquanto X[I] ≤ PIVO faça
                I ← I +1
            Fimenquanto;
            Enquanto X[J] > PIVO faça
                J ← J - 1;
            Fimenquanto;
            Se I < J
                Então AUX ← X[I];
                X[I] ← X[J];
                X[J] ← AUX;
            Fimse;
            I ← I +1;
            J ← j -1;
        Fimenquanto;
        Se J > IniVet
            Então
                QuickSort (X, IniVet, J);
            fimse;
        se I < FimVet
            Então
                Quicksort (X, J+1, FimVet);
            fimse;
    fim.

```

Você sabia

Existem alguns sites que simulam os métodos de ordenação de vetores e a maioria dos métodos que vimos neste capítulo estão representados nestes sites. Verifique o site <<https://www.toptal.com/developers/sorting-algorithms/>>, de David R. Martin. Basta clicar no botão Play All e você poderá perceber nos gráficos a eficiência entre os métodos apresentados.

9.2 Pesquisa em vetores

Quando se trabalha com grandes quantidades de dados em um vetor, fica difícil a localização de um determinado elemento de forma rápida, mesmo que o vetor esteja classificado. Para solucionar este tipo de problema, pode-se fazer uso de pesquisas por meio de algoritmos específicos. Veremos aqui métodos clássicos para efetuar pesquisa em vetores: pesquisa sequencial e pesquisa binária.

9.2.1 Método de pesquisa sequencial

Este método é bastante simples: verifica os valores em sequência. Por isto é bastante lento, não sendo indicado para grandes volumes de dados, porém pode vir a ser eficiente quando o conteúdo do vetor estiver desordenado.

× Semântica do método de pesquisa sequencial

Este método consiste em efetuar a busca da informação desejada a partir do primeiro elemento sequencialmente até o último. Localizando a informação no caminho, a mesma é apresentada.

× Sintaxe do método de pesquisa sequencial

O algoritmo faz uso de um vetor de 100 posições e utiliza a variável *chave*, que indica o valor que se deseja localizar no vetor. O *flag* OK sinaliza quando o valor de *chave* for encontrado, o que melhora a performance no método, encerrando a busca.

```

Inicio // pesquisa sequencial
  tipo VET = vetor [1:100] inteiro;
  V : VET;
  CHAVE, I, OK : inteiro;
  leia (V);
  leia (CHAVE); //valor a ser pesquisado
  OK ← 0; //flag determinando que o valor não foi encontrado
  I ← 1;
  enquanto I ≤ 100 e OK = 0 faça
    se V[I] = CHAVE
      então OK ← 1; //flag determinando que o valor foi encontrado
    fimse;
  I ← I + 1;
  fimenquanto;
  se OK = 1
    então imprima ("chave encontrada");
    senão imprima ("chave não encontrada");
  fimse;
fim.

```

9.2.2 Método de pesquisa binária

O método de pesquisa binária em vetor é, em média, mais rápido do que o anterior, porém exige que o vetor esteja previamente classificado. Veremos sua lógica e algoritmo na sequência.

- × **Semântica do método de pesquisa binária**

Este método divide o vetor em duas partes e procura saber se a informação desejada está acima ou abaixo da linha de divisão. Lembrando que o conteúdo do vetor está ordenado, então se o valor estiver acima, por exemplo, toda a metade abaixo é desprezada. Em seguida, se a informação não foi encontrada, é novamente dividida em duas partes, e verifica-se se aquela informação está acima ou abaixo, e assim executa até encontrar ou não a informação desejada.

Pelo fato de dividir sempre em duas partes o volume de dados é que o método recebe a denominação de pesquisa binária.

- × **Sintaxe do método de pesquisa binária**

Para este algoritmo novamente o vetor tem 100 elementos e o valor que se deseja encontrar está na variável *chave*. Já as variáveis *inic*,

Estrutura de Dados

final e *metade* representam os índices do vetor que indicam a posição inicial, final e metade, respectivamente, que serão utilizadas para a divisão do vetor em duas partes (binário). Segue o algoritmo.

Inicio // método binário

```
tipo VET = vetor [1:100] inteiro;
V : VET;
CHAVE, INIC, METADE, FINAL : inteiro;
leia (V);
leia (CHAVE); //valor a ser pesquisado
INIC ← 1; //indica início do vetor
FINAL ← 100; //indica final do vetor
repita
    METADE ← int(INIC+FINAL)/2; //indica posição da divisão do vetor
    se CHAVE < V[METADE]
        então FINAL ← METADE - 1; //pega a primeira parte do vetor
        senão INIC ← METADE + 1; //pega a segunda parte do vetor
    fimse;
até V[METADE] = CHAVE ou INIC > FINAL;
se V[METADE] = CHAVE
    então imprima ("chave encontrada");
    senão imprima ("chave não encontrada");
fimse;
fim.
```

Síntese

Este capítulo aborda o conteúdo de classificação e pesquisa em vetores, que é a ordenação do conteúdo de um vetor em ordem crescente ou decrescente e a busca por algum dado no vetor de informações, respectivamente. Estas atividades são realizadas constantemente nas gigantes bases de dados disponíveis, por isto a importância de se conhecer os algoritmos clássicos que já realizam estas atividades. Nas bases de dados que utilizamos diariamente, como pesquisas em máquinas de buscas na internet e redes sociais e a anterior organização destes imensos bancos de dados, algoritmos e estruturas de dados mais robustas são utilizados e é válida uma pesquisa nestes algoritmos que primam pela eficiência e eficácia.

Atividades

1. Substitua os comandos Enquanto por Para, onde for possível, no método de ordenação Bolha.
2. Substitua os comandos Enquanto por Para, onde for possível, no método de ordenação Seleção Simples.
3. Faça o teste de mesa da ordenação do vetor a seguir utilizando os 4 métodos vistos neste capítulo e identifique a eficiência de cada um dos métodos de ordenação aplicados.

S	A	U	D	E
1	2	3	4	5

4. Depois de ordenado o vetor da atividade 3, aplique o teste de mesa dos métodos de pesquisa para encontrar as letras D e Z. Identifique a eficiência de cada um dos métodos de pesquisa aplicados.

Aplique também o método de pesquisa sequencial no vetor da atividade 3 sem ordenar, buscando as letras D e Z.

Sugestão

Como sugestão, procure programar os algoritmos propostos em VisuAlg ou C, para que você entenda a lógica de cada método. Estes algoritmos são clássicos e conhecer o funcionamento deles é importante para profissionais da área de TI. Avaliações de Enade e concursos específicos para profissionais da área abordam este conhecimento.

10

Complexidade de Algoritmos

SE VOCÊ CHEGOU até este capítulo do livro e obteve um bom entendimento do conteúdo apresentado, com certeza, em se tratando de algoritmos e estruturas de dados, você já sabe fazer bem. Este capítulo tem uma proposta mais ousada: fazer um algoritmo ótimo! Isto porque, em alguns casos, não é suficiente fazer um programa funcionar retornando o resultado correto, é necessário fazê-lo funcionar de forma eficiente.

UMA REAÇÃO BASTANTE comum é quando, ao verificar que um dado aplicativo está muito lento para processar, a pessoa logo pensa em adquirir um computador mais rápido. Mas estudos mostram que, para obter um ganho maior de processamento de um dado aplicativo, é preciso buscar algoritmos mais eficientes. Um algoritmo eficiente, mesmo rodando em uma máquina lenta, quase sempre acaba derrotando um algoritmo sem eficiência rodando em uma máquina rápida.

PARA GARANTIR o desenvolvimento de algoritmos eficiente, desde a sua concepção, é importante conhecer sobre *análise de complexidade de algoritmos*.

Objetivos de aprendizagem:

1. compreender os conceitos de complexidade de algoritmos;
2. entender como são os cálculos de complexidade;
3. conhecer algumas aplicações práticas de complexidade de algoritmos.

10.1 Conceito de complexidade de algoritmos

Para podermos discutir o conceito de complexidade de algoritmos, precisamos deixar claras as definições de eficiência e eficácia. Os termos eficiência e eficácia são semelhantes, mas não são sinônimos.

10.1.1 Eficiência e eficácia

A eficiência é o ato de “fazer as coisas de maneira certa”, e a eficácia é “saber o que fazer/fazer a coisa certa”. A eficiência é quando uma pessoa age com perfeição na realização de um determinado trabalho. Já a eficácia abrange um plano mais amplo, não se limitando apenas no cumprimento de um trabalho, mas sim na resolução total de uma situação. Pode-se dizer que a eficácia é uma consequência da eficiência (ORTIZ, s.d.).

Considerando o que comentamos anteriormente, seu algoritmo é eficiente se funciona, retorna o resultado correto e de forma ótima. Esta “forma ótima” pode estar relacionada a gastar o mínimo de recursos possíveis para conseguir um bom resultado, por exemplo.

Um algoritmo feito com eficiência, mas sem eficácia, para o cliente/usuário ou sua organização, passa a ser inútil, pois não irá agregar valor, por mais bem feito que tenha sido construído o algoritmo. Então, seu algoritmo eficiente será eficaz se conseguir atender à expectativa do cliente/usuário, retornando em um benefício para ele e/ou sua organização.

10.1.2 Eficiência e corretude de um algoritmo

Um algoritmo é considerado correto quando atende a sua especificação, isto é, se para os dados de entrada válidos ele fornece como resultado a

saída esperada. Os testes e simulações utilizando diversos valores de entrada podem verificar a corretude de um algoritmo. Mas é importante salientar que estes testes garantem somente que o algoritmo funcione para os casos de teste verificados.

Quando a quantidade de valores possíveis de entrada for muito grande, torna-se impraticável o uso de casos de teste para garantir a corretude do algoritmo. Imagine o algoritmo de ordenação de uma lista com 10 posições. Existem 3.628.800 possibilidades de entradas diferentes para estas 10 posições. É possível perceber que se torna inviável aplicar os casos de teste para todas estas possibilidades. Os testes e simulações podem ser utilizados para garantir a corretude de algoritmos que têm um número pequeno de casos possíveis de entrada. Também são utilizados para descobrir e corrigir erros nas primeiras versões do algoritmo.

Já a eficiência de um algoritmo é avaliada em função do espaço de memória utilizado e tempo de execução do algoritmo. É importante citar que as estruturas de dados utilizadas no algoritmo irão determinar o espaço de memória ocupado durante a execução. Então também é preciso estar atento não só para a quantidade de dados, mas também para a forma com que estes dados serão manipulados pelo algoritmo, a estrutura de dados em si. Como o tempo de execução do algoritmo depende do número de operações executadas, a eficiência está diretamente relacionada a forma de implementação do algoritmo.

10.1.3 Análise de complexidade de algoritmos

A complexidade de um algoritmo está relacionada ao grau de esforço envolvido na solução de determinado problema. De uma forma simples, *complexidade de algoritmos* é a quantidade de trabalho necessária para executar uma tarefa, dando uma ideia do esforço computacional demandado pelo algoritmo implementado. Este trabalho envolve as funções fundamentais que o algoritmo é capaz de fazer, o volume de dados processado e a maneira como o algoritmo chega ao resultado. Entre as funções que um algoritmo é capaz de fazer, podemos citar o acesso aos dados, a inserção de novos dados, a remoção de dados, que são exemplos de funções bastante comuns na computação. Já o volume de dados refere-se à quantidade de elementos que são processados.

Para se definir a complexidade de um algoritmo, é necessário estabelecer uma medida que expresse sua eficiência, isto é, a eficiência precisa ser mensurável. Normalmente, é comum se medir a eficiência de algoritmos pelo seu “tempo” de execução ou pela quantidade de “memória” que ele utiliza para resolver o problema. A soma desses dois fatores é chamada de “custo” do algoritmo. Logo:

TEMPO + MEMÓRIA = CUSTO DO ALGORITMO

Porém, no caso da complexidade de um algoritmo, é preciso considerar quanto tempo e memória esse algoritmo gasta de acordo com o tamanho da entrada de dados. Então podemos dizer, de uma forma simplista, que:

TEMPO + MEMÓRIA + ENTRADA DE DADOS = COMPLEXIDADE DO ALGORITMO

Considerando o tempo de execução de um algoritmo, a primeira solução que vem à mente é: se um algoritmo resolver um problema em menos tempo do que o outro, logo ele será mais eficiente. Porém, um determinado algoritmo pode estar executando em um computador com hardware melhor do que o outro algoritmo, dando a falsa impressão de que ele é mais eficiente. Então, tempo absoluto não é uma boa medida de complexidade.

Em análise de complexidade de algoritmos, deve-se contabilizar o número de operações consideradas relevantes realizadas pelo algoritmo, e expressar esse número como uma função de “n”. Tais operações podem ser comparações, operações aritméticas, manipulação de dados, entre outras operações computacionais.

O número de operações realizadas por um determinado algoritmo depende também das entradas a ele fornecidas. O maior número de operações utilizadas para atender qualquer entrada de tamanho “n” é o que consideramos o “pior caso”, e este é o que merece maior atenção. Ainda há o “caso médio”, no qual as entradas estariam em um ponto de equilíbrio, e o “melhor caso”, que obviamente realiza o menor número de operações.

Se compararmos as funções de um algoritmo A e de um algoritmo B e percebermos que a função do algoritmo A cresce com “n” (considerado

um crescimento linear), ao passo que a função do algoritmo B cresce com “ n^2 ” (considerado um crescimento quadrático). Um crescimento quadrático é considerado pior que um crescimento linear. Dessa maneira, daremos preferência ao algoritmo A em relação ao algoritmo B, já que o algoritmo A (linear) tem uma complexidade menor do que o algoritmo B (quadrático).

10.2 Cálculo da complexidade de um algoritmo

Já sabemos que a complexidade está relacionada à quantidade de dados que serão processados, isto é, depende do tamanho da entrada de dados. Por exemplo, o número de operações executadas para encontrar o último registro em uma lista contendo 1.000 registros deve ser maior do que para uma lista com apenas 10 registros. O valor da entrada também pode determinar o esforço computacional. Por exemplo, para fazer uma busca linear em uma lista não-ordenada, o número de comparações varia em função de o valor procurado estar no primeiro ou no último registro da lista. Outro fator que influencia é a configuração dos dados, colocar em ordem uma lista que já está ordenada em pequenos trechos pode ser menos trabalhoso do que ordenar uma lista totalmente desordenada.

É possível estabelecer a complexidade medindo como o algoritmo se comporta quando ele precisa manipular, por exemplo, 10, 100 e 1000 elementos. Se ele consegue fazer em uma operação nos três casos, o algoritmo é constante, se cada um precisa 10 vezes mais operações, ele é linear e se precisa do quadrado do número de elementos, obviamente é quadrático. Considerando apenas esta comparação simples, não é difícil de notar como isto pode fazer uma enorme diferença de desempenho quando se tem grandes volumes de dados. Em algoritmos de altíssima complexidade, até pequenos volumes podem ter um desempenho bastante ruim.

Voltando no exemplo de listas lineares, a complexidade neste caso está relacionada ao tamanho da lista (número de registros da lista). Considerando que “ m ” indica o número de registros da lista, podemos dizer que a complexidade será uma função de “ m ”. Porém, conforme vimos, os valores e a configuração da lista também interferem no processo, então somente uma função não será suficiente para descrever todos os casos. Teríamos que deter-

minar uma função para cada caso. Este fato ocorre praticamente em todas as situações nas quais necessitamos calcular a complexidade do algoritmo, então os estudos reduziram a alguns casos especiais que são listados a seguir:

- a) **pior caso** – é caracterizado por entradas que resultam em um maior crescimento do número de operações conforme se aumenta o valor de “m”;
- b) **melhor caso** – é caracterizado por entradas que resultam em um menor crescimento do número de operações com o aumento do valor de “m”;
- c) **caso médio** – é quando se considera todas as entradas possíveis e as respectivas probabilidades de ocorrência. Esta categoria retrata o comportamento médio do algoritmo.

Neste momento, é possível dizer que somente a análise da complexidade de algoritmos permite a comparação de algoritmos equivalentes, que foram desenvolvidos para resolver o mesmo problema. E as funções de complexidade são obtidas utilizando ferramentas de análise matemática. Existem casos em que as funções são tão complexas que permanecem pendentes.

10.2.1 O cálculo de complexidade

Segundo Cormen et al (2002), uma forma direta de calcular a complexidade de um algoritmo seria encontrando a fórmula que resulte no número exato de operações realizadas pelo algoritmo para chegar ao resultado. Por exemplo, considere o algoritmo a seguir:

```
Para I de 0 até N-1 faça  
    Imprima (I);  
Fimpara;
```

Podemos calcular o tempo gasto de processamento para este trecho de algoritmo utilizando a fórmula:

$$T(N) = N * (\text{tempo da comparação interna de } I < N) + N * (\text{tempo do incremento de } I) + N * (\text{tempo da impressão de } I)$$

Porém, é muito trabalhoso montar uma fórmula precisa para todos os algoritmos e geralmente não é necessário. Por exemplo, suponha que

um algoritmo tenha a seguinte fórmula para calcular o tempo gasto de processamento:

$$T(N) = 10 \cdot N^2 + 137 \cdot N - 15$$

É possível verificar nesta fórmula que o termo linear ($137 \cdot N$) será dominado pelo termo quadrático ($10 \cdot N^2$) para qualquer situação em que $N \geq 14$. Quando $N > 1.000$, a contribuição do termo linear será de 1% do total. Quando $N \geq 1.000.000$, o termo linear se tornará irrelevante para o cálculo do tempo gasto de processamento. Da mesma forma, os fatores multiplicativos constantes (o 10 do termo $10 \cdot N^2$) não fazem diferença, pois não importa se $T(N) = N^2$ ou $T(N) = 10.000 \cdot N^2$, é possível verificar que, quando dobrarmos o valor de N o tempo gasto, continuará a ser multiplicado por 4 da mesma forma (^2 do termo $10 \cdot N^2$) (CORMEN et al., 2012).

As diferenças de eficiência podem ser consideradas “irrelevantes” para um pequeno número de elementos processados, ou “crescer proporcionalmente” com o número de elementos processados, tendendo a infinito. Assim, a forma mais divulgada para trabalhar a complexidade de tempo e espaço na complexidade de algoritmos ignora fatores constantes e os termos que crescem mais devagar.

A esta característica chamamos de “comportamento assintótico” ou “complexidade assintótica”. O crescimento assintótico representa a velocidade com que uma função tende ao infinito. Existem diferentes formas de se fazer análise assintótica de um algoritmo. Pode ser O-grande, Teta-grande e Ômega-grande. A mais conhecida, ou mais utilizada, é a chamada de “Notação O-grande” (leia-se ó-grande).

O O-grande é uma maneira de dar um limite superior para o tempo gasto por um algoritmo. O Ômega-grande define limites inferiores para o tempo gasto por um algoritmo. Já o Teta-grande leva em consideração os limites justos na complexidade assintótica. Como temos o risco de definir limites superiores e inferiores de forma não precisa, se soubermos que o limite é preciso, isto é, O(g) e Omega(g) ($\Omega(g)$) valores justos, podemos deixar isto claro para o observador escrevendo o valor de Teta(g) ($\leftarrow(g)$).

Como a notação O-grande é a mais utilizada, comentaremos sobre ela.

10.2.2 Notação O-grande

Considerando que é mais importante saber que o número de operações executadas em um algoritmo dobra se dobrarmos o valor de “m”, do que para “m” valendo 100, serão executadas 400 operações, algumas funções matemáticas elementares são utilizadas como referência para classes de funções de complexidade que consideram este valor seu limite superior (lembmando que O-grande define os limites superiores).

Quando dizemos que uma função de complexidade $f(m)$ é da ordem de m^2 , significa que as duas funções $f(m)$ e m^2 tendem ao infinito com a mesma velocidade ou que têm o mesmo comportamento assintótico. Esta situação é representada por:

$$f(m) = O(m^2)$$

Existindo outro algoritmos para o mesmo problema com função de complexidade $f_1(m) = O(m)$, podemos comparar $f(m)$ e $f_1(m)$ e estaremos comparando a eficiência dos dois algoritmos ($O(m^2)$ e $O(m)$). Em f_1 o tempo de execução é linear ($O(m)$), isto é, dobrando o tamanho da entrada dobra também o tempo de execução; em $f(m)$, o tempo é quadrático ($O(m^2)$), o que significa que, dobrando o tamanho da entrada, o tempo de execução quadruplica.

Segue o quadro 10.1, bastante utilizado e que resume alguns resultados das funções de complexidade, considerando m o tamanho da entrada, conforme relatado anteriormente no exemplo da lista.

Quadro 10.1 – Resumo de resultados de funções de complexidade (m=tamanho da entrada)

Função	Descrição	Significado
1	Tempo constante	O número de operações é o mesmo para qualquer tamanho de entrada.
m	Tempo linear	Se m dobra o número de operações também dobra.
m^2	Tempo quadrático	Se m dobra o número de operações quadruplica.

Função	Descrição	Significado
$\lg m$	Tempo logarítmico	Se m dobra o número de operações aumenta de uma constante.
$m\lg m$	Tempo $m\lg m$	Se m dobra o número de operações ultrapassa o dobro do tempo da entrada de tamanho m.
2^m	Tempo exponencial	Se m dobra o número de operações é elevado ao quadro.

Fonte: Adaptado de Ziviani (2010).

É importante citar que os resultados expressos em notação O, que trabalha com o limite superior, devem ser interpretados com cuidado, uma vez que indicam que o tempo de execução do algoritmo é proporcional a um determinado valor, ou que nunca supera determinado valor. Em alguns casos, o tempo de execução pode ser bastante inferior ao valor indicado, ou até o pior caso pode nunca ocorrer.

Os cálculos matemáticos utilizados para definir a complexidade de algoritmos são bastante complexos e não caberia neste material descrevê-los. Caso o leitor tenha interesse em aprofundar o conhecimento, os materiais citados nas referências deste capítulo são elucidativos e indicados. Também o livro *Fundamentos Matemáticos para Ciência da Computação*, escrito por Judith L. Gersting, da Editora LTC, pode ser citado como fonte de consulta.

10.2.3 Exemplo de cálculo de complexidade

Vamos utilizar como exemplo uma busca linear em uma lista como m elementos (tamanho m). A operação preponderante em uma busca linear é a comparação da chave x (valor que desejamos encontrar na lista) com os m valores da lista.

Para este exemplo, serão considerados dois casos:

$$W(m) = \text{número de comparações no pior caso};$$

$$A(m) = \text{número de comparações no caso médio}.$$

Estrutura de Dados

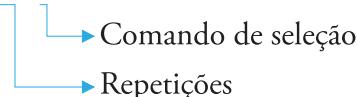
O algoritmo de busca que será utilizado é o que segue e a função de complexidade deve indicar o número de vezes que é testada a condição $(A[J] \neq X)$.

```
J ← 1;  
Enquanto  $(A[J] \neq X)$  e  $(X < M)$  faça  
    Se  $A[J] \neq X$   
        Então achou ← falso;  
        Senão achou ← verdadeiro;  
        posição ← J;  
    Fimse;  
    J ← J + 1;  
Fimenquanto;
```

No pior caso, quando o valor de X não existe na lista e precisamos verificá-la até o final para não encontrá-lo, temos que a condição $A[J] \neq X$ ocorre como condição de controle da repetição e mais uma vez no comando Se. Já a outra condição que controla a repetição $J < M$, reduz o número de repetições a $M - 1$. Desta forma, se $A[J] \neq X$ tiver sempre o valor falso, a repetição será executada $M - 1$ vezes, porque J é inicializado com 1. Entretanto, a condição que controla a repetição é verificada m vezes, que são as $M - 1$ vezes que a repetição é executada mais uma vez quando J atinge o valor de m e $J < M$ torna-se falso.

Logo temos que:

$$W(m) = m + 1$$



Por fim, temos que: $W(m) = O(m)$ no pior caso.

No caso médio, precisamos considerar que na busca por um valor X na lista é possível que X seja encontrado na primeira, segunda ou terceira posição do meio da lista, última posição da lista, ou ainda que X não se encontra na lista (pior caso). Serão, portanto, $M + 1$ casos diferentes de entradas possíveis (I entradas possíveis). Para cada um destes casos, é preciso associar um

número de probabilidade de que a entrada ocorra. Considerando que todas as entradas sejam igualmente prováveis, temos como probabilidade $1 / (M + 1)$ para cada entrada I.

Considerando C_i o número de comparações executadas quando ocorre a entrada I. O teste da condição $A[J] <> X$, que controla a repetição, será executado para cada valor de J, de 1 até I, quando $A[I] <> X$ for falso. Então, serão I comparações na repetição mais uma comparação fora da repetição. Se X não estiver na lista, serão efetuadas $M + 1$ comparações.

Concluindo, $A(m) = O(m)$, para o caso médio.

10.3 Aplicações práticas da complexidade de algoritmos

De forma geral, a análise de complexidade de algoritmos busca respostas para perguntas como:

- ✗ este algoritmo resolve o meu problema?
- ✗ quanto tempo o algoritmo consome para processar uma entrada de tamanho “m”?

Conforme visto neste capítulo, as respostas a essas perguntas podem ser um pouco complexas, algo como o consumo de tempo é proporcional a $n^2 \log n$, no pior caso.

Além disso, a análise de complexidade de algoritmos estuda certos paradigmas da computação, como divisão-e-conquista, programação dinâmica, algoritmo da gula, busca local, busca por aproximação, entre outros, que se mostram úteis na criação de diversos algoritmos para solução de problemas computacionais complexos.

De forma mais prática, conhecer o método de cálculo de complexidade de algoritmos e entender seu resultado é importante para definirmos a escolha de certos algoritmos em detrimento de outros. Por exemplo, considerando os diversos algoritmos de busca existentes podemos criar o quadro 10.2 a seguir.

Estrutura de Dados

Quadro 10.2 – Comparando algoritmos de busca usando a complexidade de algoritmos

Algoritmo	Descrição	Complexidade		
		Melhor caso	Caso médio	Pior caso
Busca binária	Algoritmo de busca em vetores que segue o paradigma de divisão e conquista	$O(1)$	$O(\log n)$, em que n é o número de elementos	$O(\log n)$, Em que n é o número de elementos
Busca sequencial	Algoritmo de busca em vetores ou listas de forma sequencial, elemento por elemento.	$O(1)$	$O(n)$, em que n é o número de elementos	$O(n)$, em que n é o número de elementos
Busca em profundidade	Algoritmo para realizar busca ou travessia em uma estrutura de dados do tipo árvore ou grafo em profundidade.	$O(1)$	$O(V + A)$, em que V são os vértices e A as arestas	$O(V + A)$, em que V são os vértices e A as arestas
Busca em largura	Algoritmo para realizar busca ou travessia em uma estrutura de dados do tipo árvore ou grafo em largura.	$O(1)$	$O(V + A)$, em que V são os vértices e A as arestas	$O(V + A)$, em que V são os vértices e A as arestas

Fonte: Elaborado pelo autor.

Por meio de quadros comparativos de complexidade entre algoritmos que desempenham a mesma função, é possível escolher o algoritmo mais eficiente para a situação-problema a ser resolvida. Algoritmos clássicos e de uso comum, como os de busca e ordenação, já apresentam seus resultados em relação à complexidade, o que permite uma análise mais rápida, lembrando

que é necessário verificar a estrutura de dados utilizada e o padrão da entrada de dados para saber se está adequada ao uso desejado.

Agora, vamos comparar algoritmos que resolvem a série de Fibonacci, cujos 10 primeiros termos são: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. A sequência pode ser definida recursivamente como:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Dado o valor de n , deseja-se obter o “n-ésimo” elemento da sequência. Serão apresentados dois algoritmos e a complexidade de cada um.

Seja a função Fibo1(N) que calcula o “n-ésimo” elemento da sequência de Fibonacci.

```

Entrada - valor de n
Saída - o “n-ésimo” elemento da sequência de Fibonacci
Função Fibol (N:inteiro)
    Se N = 0
        Então retorne (0);
    Senão Se N = 1
        Então retorne (1);
        Senão retorne(Fibol(N - 1) + Fibol(N - 2));
    Fimse;
Fimse;
fimFibol;
```

A complexidade deste algoritmo é de $O(2^n)$.

Experimente executar este algoritmo para $n = 100$. Considerando que uma operação leve um picosegundo, 2^{100} operações levariam $3 * 10^{13}$ anos, o que significam 30.000.000.000.000 anos!

O algoritmo Fibo2(N) também resolve a série de Fibonacci, porém utilizando outra lógica de solução:

```

Função Fibol (N:inteiro)
    Penultimo, ultimo, atual : inteiro;
    Se N = 0
```

Estrutura de Dados

```
Então retorne (0);
Senão Se N = 1
    Então  retorne (1);
    Senão penúltimo ← 0;
        Último ← 1
        Para I de 2 até n faça
            Atual ← penúltimo + último;
            Penúltimo ← último;
            Último ← atual;
        Fimpara;
        Retorne (atual);
    Fimse;
Fimse;
fimFibo2;
```

Com esta lógica, a complexidade passou para $O(n)$.

Você deve ter percebido que o algoritmo Fib01 utiliza uma solução recursiva, enquanto a Fib02 utiliza uma solução linear.

Síntese

Neste capítulo vimos que, no desenvolvimento de algoritmos, é importante conhecer sua eficiência e não somente se o resultado está correto. Para avaliar a eficiência de um algoritmo normalmente mede-se o tempo de execução e o espaço de memória ocupado pelo algoritmo durante a execução do programa associado a ele. O tempo de execução varia com a entrada de dados. A análise de complexidade de algoritmos auxilia na avaliação destes casos. O cálculo da complexidade de algoritmos é trabalhoso e complexo, mas seu entendimento permite identificar algoritmos mais eficientes em detrimento de outros na solução do mesmo problema, permitindo uma escolha adequada.

Atividades

1. Realize uma pesquisa e identifique a complexidade dos algoritmos clássicos de ordenação de vetores.
2. Por que normalmente damos atenção apenas para o pior caso na complexidade de algoritmos? Justifique.

3. Considere que você tenha dois algoritmos A e B, que se destinam a resolver um mesmo problema. O algoritmo A tem complexidade $O(n^5)$ e o algoritmo B $O(2^n)$. Você utilizaria o algoritmo B ao invés do A? Em qual caso?
4. Uma empresa necessita comprar um aplicativo de software para incorporar em seu sistema. O setor de TI está avaliando, entre outras características, a complexidade do algoritmo utilizado nos aplicativos de cada empresa fornecedora, utilizando a complexidade assintótica na notação O-grande. No quadro a seguir estão relacionadas as complexidades de cada um dos participantes da concorrência:

Empresa	Complexidade
Alfa	$O(n^{20})$
Sigma	$O(1)$
Delta	$O(n \log n)$
Gama	$O(n!)$
Omega	$O(2^n)$

Qual foi a empresa que ficou em segundo lugar na lista do setor de TI com relação à complexidade do algoritmo empregado? Explique seu cálculo.

Conclusão

Parabéns por ter chegado até aqui. A partir de agora, novos desafios o esperam a cada disciplina e tenho certeza de que, a cada desafio vencido, você se aproxima do seu objetivo, que é o mercado de trabalho em TI. Este material proporcionou a você uma experiência aprofundada em diversas estruturas abstratas de dados, verdadeiras obras de arte da computação, a exemplos das árvores, algoritmos de percurso em grafos, estrutura de alocação dinâmica e complexidade de algoritmos, entre outros temas abordados aqui. Conteúdo profundo e que exige muito de nossa capacidade de abstração, pois são estruturas que não existem fisicamente, são padrões de comportamento que impomos aos dados por meio dos algoritmos. Quem sabe por isto seja tão complexo de ser entendido!

Procurou-se apresentar o conteúdo de maneira objetiva, complementando textos com ilustrações e tabelas, especialmente para ilustrar procedimentos mais complexos de serem visualizados a partir, unicamente, da leitura de um texto descritivo ou narrativo. Também os algoritmos apresentados procuram ser genéricos, para que você possa transpor facilmente para linguagem de programação. Assim a sua compreensão será mais fácil, impulsionando-o para a realização e a compreensão aprofundada das implementações destas estruturas de dados.

Ao ler este livro, foi dado um grande passo. Continue caminhando!

Gabarito

1. Estrutura de Dados Homogêneos e Heterogêneos

1.

```
inicio
    Tipo VET = VETOR [1:10] real;
    A: VET;
    I: inteiro;
    SOMA, MEDIA, DIF: real;
    SOMA ← 0;
    Para I de 1 até 10 faça
        Leia (A[I]);
        SOMA ← SOMA + A[I];
    fimpara;
    MEDIA ← SOMA / 10;
    Imprima (MEDIA);
    Para I de 1 até 10 faça
        imprima (A[I]);
        DIF ← A[I] - MEDIA;
        imprima(DIF);
    fimpara;
fim.
```

2.

```
inicio
    Tipo VET = VETOR [1:10] inteiro;
    Tipo VET1 = VETOR [0:10] inteiro;
    GAB: VET;
    FREQ: VET1;
    I, ALUNO, NOTA, RESP, MAIOR, NOTAMAIOR: inteiro;
    Para I de 1 até 10 faça
        Leia (GAB[I]);
    fimpara;
    Para I de 0 até 10 faça
        FREQ [I] ← 0; //para somar o número de vezes que cada nota de
0 a 10 apareceu
    fimpara;
    (TOT, APROV) ← 0;
    leia (ALUNO);
    enquanto ALUNO <> 9999 faça
        NOTA <- 0; // calcula nota do aluno
        para I de 1 até 10 faça
```

```

leia (RESP);
se RESP = GAB[I]
    então NOTA ← NOTA + 1;
fimse;
fimpara;
imprima (ALUNO, "tirou nota", NOTA);
FREQ [NOTA] ← FREQ [NOTA] + 1; //insere no vetor de
frequência
Se NOTA >= 60 //calcular o % de aprovação
    Então APROV ← APROV + 1;
Fimse;
TOT ← TOT + 1;
Leia (ALUNO);
Fim enquanto;
PERC ← (APROV / TOT) * 100;
Imprima ("percentual de aprovação foi de ", PERC,
"%");
MAIOR ← 0; //algoritmo de maior usando o vetor FREQ
NOTAMAIOR ← 0;
Para I de 0 até 10 faça
    Se FREQ[I] > MAIOR
        Então MAIOR ← FREQ[I];
        NOTAMAIOR ← I; //guarda a nota que tem maior frequ-
ênciá até o momento
    Fimse;
    Fimpara;
    Imprima ("a nota que teve maior frequência abso-
luta foi ", NOTAMAIOR);
fim.

```

3.**Inicio**

```

tipo M = matriz [1:8, 1:8] inteiro;
tipo V = vetor [0:6] inteiro;
tipo V1 = vetor [0:6] caractere [20];
TAB:M;
PECA: V;
NOME: V1;
I,J, SOMA:inteiro;
Para I de 0 até 6 faça
    PECA[I] ← 0; //zera a quantidade de cada peça
    Caso I //associa o nome da peça
        0: NOME[I] ← "ausência de peça";

```

Estrutura de Dados

```
1: NOME[I] ← "peão";
2: NOME[I] ← "cavalo";
3: NOME[I] ← "torre";
4: NOME[I] ← "bispo";
5: NOME[I] ← "rei";
Senão : NOME[I] <- "rainha";
Fimcaso;
para I de 1 até 8 faca
    para J de 1 até 8 faca
        leia (TAB[I,J]; //lê o tabuleiro
        PECA[TAB[I,J]] ← PECA[TAB[I,J]] + 1; //usa o
valor do tabuleiro como índice do vetor de quantidade de cada peça
    fimpara;
    fimpara;
    para I de 0 até 6 faç�
        imprima ("quantidade de ", NOME[I], "é de ",
PECA[I]);
    fimpara;
fim.
```

4.

```
inicio
    Tipo DATA = registro D,M,A: inteiro;
        fímdata;
    Tipo REG1 = registro MATR : inteiro;
                    NOME : caractere [40];
                    NASC ; DATA;
                    SEXO : caractere [1];
    fímreg1;
    tipo REG2 = registro NOME : caractere [40];
                    IDADE: inteiro;
    fímreg2;
    SEGURADO : REG1;
    HOMENS, MULHERES: REG2;
    ATUAL: DATA;
    IDADE: inteiro;
    Leia (ATUAL.D, ATUAL.M, ATUAL.A);
    Leia (SEGURADO.MATR);
    Enquanto SEGURADO.MATR <> 0 façå
        Leia (SEGURADO.NOME,
                SEGURADO.NASC.D,
                SEGURADO.NASC.M,
```

```

SEGURADO.NASC.A,
SEGURADO.SEXO);
IDADE ← (ATUAL.A - SEGURADO.NASC.A) +
(ATUAL.M - SEGURADO.NASC.M);
Se SEGURADO.SEXO = "M"
Então HOMENS.NOME ← SEGURADO.NOME;
HOMENS.IDADE ← IDADE;
Senão MULHERES.NOME ← SEGURADO.NOME;
MULHERES.IDADE ← IDADE;
Fimse;
Leia (SEGURADO.MATR);
Fimenquanto;
Fim.

```

2. Estruturas de Dados Elementares

Para resolver os algoritmos propostos nas atividades, serão utilizadas as funções de empilhar e desempilhar descritas neste capítulo. É importante lembrarmos que para mantermos a ordem dos valores originais da Pilha é preciso utilizar uma Pilha auxiliar (PilhaAux), realizar as alterações solicitadas por cada um dos enunciados e depois voltar a ordem na Pilha original.

1.

```

algoritmo
CONST MAX = 1000
TIPO VET = VETOR[MAX] : real;
TOPO, TOPOAUX : inteiro;

procedimento empilhar(V : VET, TP: inteiro);
procedimento desempilhar (V : VET, TP: inteiro):
imprimir_pilha(V : VET, TP: inteiro);

inicio
PILHA, PILHAAUX : VET;
VALOR: real;
(TOPO, TOPOAUX) ← 1;
Para I de 1 até MAX faça
Leia (VALOR);
Empilhar (PILHA, TOPO);
Fimpara;
Para I de 1 até MAX faça

```

Estrutura de Dados

```
Desempilhar (PILHA, TOPO);
Se VALOR >= 0
    Então Empilhar (PILHAAUX, TOPOAUX);
Fimse;
Fimpara;
//empilhado em PILHAAUX os valores >=0, mas na ordem inversa da pilha
original
Para I de 1 até TOPOAUX faça //colocando Pilha na ordem
original
    Desempilhar (PILHAAUX, TOPOAUX);
    Empilhar (PILHA, TOPO);
Fimpara;
Imprimir_pilha (PILHA, TOPO);
fim //algoritmo

procedimento empilhar(V : VET, TP; inteiro)
    inicio
        se TP > MAX
            então
                imprima("PILHA CHEIA!");
                empilhar;
            fimse
            V[TP] ← VALOR;
            TP ← TP + 1;
        fim; // procedimento empilhar

procedimento desempilhar(V : VET, TP: inteiro)
    inicio
        se TP = 1
            então
                imprima("PILHA VAZIA!");
                desempilhar;
            fimse;
            VALOR ← V[TP];
            TP ← TP - 1;
        fim; // procedimento desempilhar

procedimento imprimir_pilha(V : VET, TP: inteiro)
    inicio
        J : inteiro;
        imprima("PILHA : ");
        se TP = 1
            então
                imprima("PILHA VAZIA!");
```

```

        imprimir_pilha;
senão
    para J de TP - 1 até 1 passo - 1 faça
        imprima(V[J]);
        se J = TP - 1
            então
                imprima("← TOPO");
            fimse;
            imprima( );
        fimpara;
    fimse;
fim; // procedimento imprimir_pilha

```

2.

algoritmo

```

CONST MAX = 100
TIPO VET = VETOR[MAX]: real;
TOPO, TOPOAUX: inteiro;

procedimento empilhar(V : VET, TP: inteiro);
procedimento desempilhar (V : VET, TP: inteiro);
imprimir_pilha(V : VET, TP: inteiro):

inicio
    PILHA, PILHAAUX: VET;
    VALOR: real;
    (TOPO, TOPOAUX) ← 1;
    Para I de 1 até MAX faça
        Leia (VALOR);
        Empilhar (PILHA, TOPO);
    Fimpara;
    Para I de 1 até MAX faça
        Desempilhar (PILHA, TOPO);
        empilhar (PILHAAUX, TOPOAUX);
    Fimpara;
//empilhado em PILHAAUX os valores ficam na ordem inversa da pilha
original
    Imprimir_pilha (PILHAAUX, TOPOAUX);
fim; //algoritmo

procedimento empilhar(V : VET, TP; inteiro)
    inicio
        se TP > MAX

```

Estrutura de Dados

```
        então
            imprima("PILHA CHEIA!");
            empilhar;
        fimse
        V[TP] ← VALOR;
        TP ← TP + 1;
    fim // procedimento empilhar

procedimento desempilhar(V : VET, TP: inteiro)
    inicio
        se TP = 1
            então
                imprima("PILHA VAZIA!");
                desempilhar;
            fimse;
        VALOR ← V[TP];
        TP ← TP - 1;
    fim // procedimento desempilhar

procedimento imprimir_pilha(V : VET, TP: inteiro)
    inicio
        J : inteiro;
        imprima("PILHA : ");
        se TP = 1
            então
                imprima("PILHA VAZIA!");
                imprimir_pilha;
            senão
                para J de TP - 1 até 1 passo - 1 faça
                    imprima(V[J]);
                    se J = TP - 1
                        então
                            imprima("← TOPO");
                        fimse;
                    imprima( );
                fimpara;
            fimse;
    fim // procedimento imprimir_pilha
```

3.

```
algoritmo
    CONST MAX = 500
    TIPO VET = VETOR[MAX]: inteiro;
```

```

TOPO, TOPOAUX, TOPOPAR, TOPOIMPAR: inteiro;

procedimento empilhar(V : VET, TP: inteiro);
procedimento desempilhar (V : VET, TP: inteiro);
imprimir_pilha(V : VET, TP: inteiro);

inicio
    PILHA, PILHAAUX, PILHAPAR, PILHAIMPAR: VET;
    VALOR: inteiro;
    (TOPO, TOPOPAR,TOPOIMPAR) ← 1;
    Para I de 1 até MAX faça
        Leia (VALOR);
        Empilhar (PILHA, TOPO);
        Fimpara;
    Para I de 1 até MAX faça
        Desempilhar (PILHA,TOPO);
        Se VALOR mod 2 = 0
            Então Empilhar (PILHAPAR,TOPOPAR);
            Senão Empilhar (PILHAIMPAR,TOPOIMPAR);
        Fimse;
        Fimpara;
    //empilhado em PILHAPAR e PILHAIMPAR os valores estarão na ordem
    inversa da pilha original
    TOPOAUX ← 1;
    Para I de 1 até TOPOPAR faça //colocando PilhaPar na ordem
    original
        Desempilhar (PILHAPAR,TOPOPAR);
        Empilhar (PILHAAUX, TOPOAUX);
        Fimpara;
        Imprimir_pilha (PILHAAUX,TOPOAUX);
        TOPOAUX ← 1;
        Para I de 1 até TOPOPAR faça //colocando PilhaImpar na
        ordem original
            Desempilhar (PILHAIMPAR,TOPOIMPAR);
            Empilhar (PILHAAUX, TOPOAUX);
        Fimpara;
        Imprimir_pilha (PILHAAUX,TOPOAUX);
    fim. //algoritmo

procedimento empilhar(V : VET, TP; inteiro)
    inicio
        se TP > MAX
            então
                imprima("PILHA CHEIA!");

```

Estrutura de Dados

```
        empilhar;
fimse
V[TP] ← VALOR;
TP ← TP + 1;
fim; // procedimento empilhar

procedimento desempilhar(V : VET, TP: inteiro)
inicio
    se TP = 1
        então
            imprima("PILHA VAZIA!");
            desempilhar;
    fimse;
    VALOR ← V[TP];
    TP ← TP - 1;
fim; // procedimento desempilhar

procedimento imprimir_pilha(V : VET, TP: inteiro)
inicio
    J : inteiro;
    imprima("PILHA : ");
    se TP = 1
        então
            imprima("PILHA VAZIA!");
            imprimir_pilha;
    senão
        para J de TP - 1 até 1 passo - 1 faça
            imprima(V[J]);
        se J = TP - 1
            então
                imprima("← TOPO");
        fimse;
        imprima( );
    fimpara;
fimse;
fim // procedimento imprimir_pilha
```

4.

```
algoritmo
    CONST MAX = 250;
    TIPO VET = VETOR[MAX]: caractere [70];
    TOPO, TOPOAUX: inteiro;
```

```

procedimento empilhar(V : VET, TP: inteiro);
procedimento desempilhar (V : VET, TP: inteiro);
imprimir_pilha(V : VET, TP: inteiro);

inicio
    PILHA, PILHAUX: VET;
    NOME, NOMEBASE, NOMETOPO: caractere [70];
    (TOPO, TOPOAUX) ← 1;
    Para I de 1 até MAX faça
        Leia (NOME);
        Empilhar (PILHA, TOPO);
    Fimpara;
    Desempilhar (PILHA, TOPO); //separa o nome do topo da pilha
e não coloca em PilhaAux
    NOMETOPO ← NOME;
    Para I de 2 até MAX faça //desempilha o resto da pilha e coloca
em PilhaAux
        Se I = MAX //Separa o nome da base da pilha e não coloca em
PilhaAux
            Então desempilhar (PILHA, TOPO);
            NOMEBASE ← NOME;
        Senão
            Desempilhar (PILHA,TOPO);
            Empilhar (PILHAAUX,TOPOAUX);
        Fimpara;
//monta a pilha novamente a partir de PilhaAux trocando o topo pela base
e vice-versa
    TOPO ← 1;
    NOME ← NOMETOPO;
    Empilhar (PILHA, TOPO); //empilhou a nova base
        Para I de 1 até TOPOAUX faça
            Desempilhar (PILHAUX, TOPOAUX);
            Empilhar (PILHA, TOPO);
    Fimpara;
    NOME ← NOMEBASE;
    Empilhar (PILHA, TOPO); //empilhou novo topo
    Imprimir_pilha (PILHA,TOPO);
fim; //algoritmo

procedimento empilhar(V : VET, TP; inteiro)
    inicio
        se TP > MAX
            então
                imprima("PILHA CHEIA!");

```

Estrutura de Dados

```
        empilhar;
fimse
    V[TP] ← NOME;
    TP ← TP + 1;
fim; // procedimento empilhar

procedimento desempilhar(V : VET, TP: inteiro)
    início
        se TP = 1
            então
                imprima("PILHA VAZIA!");
                desempilhar;
            fimse;
        NOME ← V[TP];
        TP ← TP - 1;
    fim; // procedimento desempilhar

procedimento imprimir_pilha(V : VET, TP: inteiro)
    início
        J : inteiro;
        imprima("PILHA : ");
        se TP = 1
            então
                imprima("PILHA VAZIA!");
                imprimir_pilha;
            senão
                para J de TP - 1 até 1 passo - 1 faça
                    imprima(V[J]);
                    se J = TP - 1
                        então
                            imprima("← TOPO");
                    fimse;
                    imprima( );
                fimpara;
            fimse;
    fim; // procedimento imprimir_pilha
```

3. Estrutura de Dados Elementares: Fila

1.

algoritmo

```

CONST MAX = 200
TIPO VET = VETOR[MAX]: caractere [40];
INÍCIO, FIM : inteiro;

procedimento enfileirar(V:VET, FI:inteiro);
            procedimento      desenfileirar
(V:VET, IN:inteiro, FI:inteiro);
            procedimento      imprimir_fila
(V:VET, IN:inteiro, FI:inteiro);

inicio
    FILA, FILAAUX: VET;
    I, INIAUX, FIMAUX: inteiro;
    INÍCIO, FIM, FIMAUX, INIAUX) ← 1;
    NOME: caractere [40];
    Para I de 1 até MAX faça
        Leia (NOME);
        Enfileirar (FILA, FIM);
        Fimpara;
    Para I de 1 até MAX faça
        Desenfileirar (FILA, INICIO, FIM);
        Enfileirar (FILAAUX, FIMAUX);
        Fimpara;
    Imprimir_fila (FILAAUX, INIAUX, FIMAUX);
fim. //algoritmo

procedimento enfileirar(V : VET, FI: inteiro)
    inicio
        se FI > MAX
            então
                imprima("FILA CHEIA!");
                enfileirar;
            fimse;
        V[FI] ← NOME;
        FI ← FI + 1;
    Fim; // procedimento enfileirar

procedimento      desenfileirar(V:VET,      IN:inteiro,
FI:inteiro)
    inicio
        se IN = FI
            então
                imprima("FILA VAZIA!");
                desenfileirar;

```

Estrutura de Dados

```
fimse;
NOME ← V[IN];
IN ← IN + 1;
Fim; //procedimento desenfileirar

procedimento imprimir_fila(V:VET, IN:inteiro, FI:inteiro)
    inicio
        J : inteiro;
        imprima("INÍCIO DA Fila : ");
        se IN = FI
            então
                imprima("FILA VAZIA!");
            senão
                para J de IN até FI - 1 faça
                    imprima(V[J]);
                fimpara;
        fimse;
        imprima("FIM DA Fila");
    fim; // procedimento imprimir_fila
```

2.

```
algoritmo
    CONST MAX = 1000
    TIPO VET = VETOR[MAX]: inteiro;
    INÍCIO, FIM : inteiro;

    procedimento enfileirar(V:VET, FI:inteiro);
        procedimento desenfileirar
            (V:VET, IN:inteiro, FI:inteiro);
            procedimento imprimir_fila
                (V:VET, IN:inteiro, FI:inteiro);

    inicio
        FILA, FILAAUX: VET;
        I, INIAUX, FIMAUX: inteiro;
        INÍCIO, FIM, FIMAUX, INIAUX) ← 1;
        VALOR: inteiro;
        Para I de 1 até MAX faça
            Leia (VALOR);
            Enfileirar (FILA, FIM);
            Fimpara;
        Para I de 1 até MAX faça
```

```

        Desenfileirar (FILA, INICIO, FIM);
        Se VALOR mod 7 <> 0
            Então
                Enfileirar (FILAUX, FIMAUX);
            Fimse;
        Fimpara
        (INICIO, FIM) ← 0;
        Para I de 1 até FIMAUX faça
            Desenfileirar (FILAUX, INIAUX, FIAUX);
            Enfileirar (FILA, FIM);
        Fimpara;
        Imprimir_fila (FILA, INICIO, FIM);
    fim. //algoritmo

procedimento enfileirar(V : VET, FI: inteiro)
    início
        se FI > MAX
            então
                imprima("FILA CHEIA!");
                enfileirar;
            fimse;
        V[FI] ← VALOR;
        FI ← FI + 1;
    Fim; // procedimento enfileirar

    procedimento desenfileirar(V:VET, IN:inteiro,
FI:inteiro)
    início
        se IN = FI
            então
                imprima("FILA VAZIA!");
                desenfileirar;
            fimse;
        VALOR ← V[IN];
        IN ← IN + 1;
    Fim; //procedimento desenfileirar

procedimento imprimir_fila(V:VET, IN:inteiro, FI:intei
ro)
    início
        J : inteiro;
        imprima("INÍCIO DA Fila : ");
        se IN = FI
            então

```

Estrutura de Dados

```
        imprima("FILA VAZIA!");
        senão
            para J de IN até FI - 1 faça
                imprima(V[J]);
            fimpara;
        fimse;
        imprima("FIM DA Fila");
fim; // procedimento imprimir_fila
```

3.

```
algoritmo
    CONST MAX = 2000
    TIPO DATA = REGISTRO D,M,A; inteiro;
    fimDATA;
    TIPO REG = REGISTRO MATR: inteiro;
                           NASC: DATA;
    fimREG;
    TIPO VET = VETOR[MAX]: REG;
    INÍCIO, FIM : inteiro;

    procedimento enfileirar(V:VET, FI:inteiro);
        procedimento           desenfileirar
(V:VET, IN:inteiro, FI:inteiro);
        procedimento           imprimir_fila
(V:VET, IN:inteiro, FI:inteiro);

inicio
    FILA, FILAAUX: VET;
    ATUAL: DATA;
    I, IDADE, INIAUX, FIMAUX: inteiro;
    INÍCIO, FIM, FIMAUX, INIAUX) ← 1;
    JOGADOR: REG;
    Leia (ATUAL.D, ATUAL.M, ATUAL.A);
    Para I de 1 até MAX faça
        Leia (JOGADOR.MATR, JOGADOR.NASC);
        Enfileirar (FILA, FIM);
    Fimpara;
    Para I de 1 até MAX faça
        Desenfileirar (FILA, INICIO, FIM);
        IDADE ← (ATUAL.A - JOGADOR.NASC.A) +
                  (ATUAL.M - JOGADOR.NASC.M);
    Se IDADE < 18
        Então
```

```

        Enfileirar (FILAUX, FIMAUX);
        Fimse;
        Fimpara
        (INICIO, FIM) ← 0;
        Para I de 1 até FIMAUX faça
            Desenfileirar (FILAUX, INIAUX, FIAUX);
            Enfileirar (FILA, FIM);
        Fimpara;
        Imprimir_fila (FILA, INICIO, FIM);
    fim. //algoritmo

procedimento enfileirar(V : VET, FI: inteiro)
    inicio
        se FI > MAX
            então
                imprima("FILA CHEIA!");
                enfileirar;
        fimse;
        V[FI].MATR ← JOGADOR.MATR;
        V[FI].NASC ← JOGADOR.NASC;
        FI ← FI + 1;
    Fim; // procedimento enfileirar

procedimento desenfileirar(V:VET,     IN:inteiro,
FI:inteiro)
    inicio
        se IN = FI
            então
                imprima("FILA VAZIA!");
                desenfileirar;
        fimse;
        JOGADOR.MATR ← V[IN].MATR;
        JOGADOR.NASC ← V[IN].NASC;
        IN ← IN + 1;
    Fim; //procedimento desenfileirar

procedimento imprimir_fila(V:VET,IN:inteiro,FI:intei
ro)
    inicio
        J : inteiro;
        imprima("INÍCIO DA Fila : ");
        se IN = FI
            então
                imprima("FILA VAZIA!");

```

```

        senão
            para J de IN até FI - 1 faça
                imprima(V[J].MATR, V[J].NASC);
            fimpara;
        fimse;
        imprima("FIM DA Fila");
fim; // procedimento imprimir_fila

```

4. Ponteiros e Alocação Dinâmica

1. Sim, pois o valor está sendo lido para o conteúdo de uma variável ponteiro, que aponta para o endereço da variável desejada.

2.

a) $*P - *Q$

Ação	Comentário
$*P - *Q$	Substituindo o significado de * na expressão.
$(\text{Valor contido no endereço de memória } P) - (\text{Valor contido no endereço de memória } Q)$	Substituindo o valor dos ponteiros P e Q na expressão.
$(\text{Valor contido no endereço de memória } &I) - (\text{Valor contido no endereço de memória } &J)$	Substituindo o significado de & na expressão
$(\text{Valor contido no endereço de memória de } I) - (\text{Valor contido no endereço de memória de } J)$	Substituindo pelo valor que está armazenado nos endereços de I e J, que foram inseridos no comando de atribuição $I \leftarrow 3$ e $J \leftarrow 5$.
$3 - 5$	Realizando o cálculo matemático.
-2	Resposta da expressão.

b) $3 - *P / (*Q + 7)$

Ação	Comentário
$3 - *P / (*Q + 7)$	Substituindo o significado de * na expressão.
$3 - (\text{Valor contido no endereço de memória } P) / ((\text{Valor contido no endereço de memória } Q) + 7)$	Substituindo o valor dos ponteiros P e Q na expressão.
$3 - 3 / (5 + 7)$	Substituindo valor de *P e *Q considerando os comandos de atribuição $I \leftarrow 3$ e $J \leftarrow 5$.
$3 - 3 / (12)$ $3 - 0,25$	Realizando o cálculo matemático considerando as prioridades das operações.
$3 - 5$	Realizando o cálculo matemático.
2,75	Resposta da expressão.

- 3.** Qual será a saída deste algoritmo, supondo que I ocupa o endereço 4094 na memória?

Início

```

P = ponteiro para inteiro;
I : inteiro;
I ← 5;
P ← &I;
Imprima (P, *P + 2, **&P, 3 * (*P), (**&P) + 4);

```

Fim.

Ação	Comentário
Imprima (P)	Conforme o enunciado, o valor do ponteiro P é o endereço ocupado pela variável I ($P \leftarrow &I$) que é 4094.
4094	

Estrutura de Dados

Ação	Comentário
Imprima (*P + 2)	Substituindo o valor de *P que indica o conteúdo do valor apontado por P ($P \leftarrow &I$) que é 5 ($I \leftarrow 5$).
Imprima (5 + 2)	
7	Resolvendo a adição, o valor impresso é 7.
Imprima (**&P)	Esta expressão não é possível de ser realizada.
erro	
Imprima (3 * (*P))	Substituindo o valor de *P que indica o conteúdo do valor apontado por P ($P \leftarrow &I$) que é 5 ($I \leftarrow 5$).
Imprima (3 * (5))	
15	Resolvendo o produto, o valor impresso é 15.
Imprima (**&P + 4)	Esta expressão não é possível de ser realizada.
erro	

Os valores impressos serão 4094, $5+2 = 7$, erro, $3 * 5 = 15$, erro, respectivamente.

4.

a) $P \leftarrow &I;$

Esta atribuição é possível, já que P é uma variável do tipo ponteiro.

b) $P \leftarrow &&I;$

Esta atribuição é possível, já que P é uma variável do tipo ponteiro.

c) $I \leftarrow (*&)J;$

Esta atribuição não é possível, pois J não é uma variável do tipo ponteiro para receber o valor de *.

d) $*\&J;$

Este comando não é permitido nas linguagens de programação. Falta uma variável (endereço de memória alocado) para receber o resultado da expressão.

e) $I \leftarrow *\&*\&J;$

Esta atribuição não é possível, pois J não é uma variável do tipo ponteiro para receber o valor de *.

f) $I \leftarrow (*P) + (3 * (*Q));$

O resultado das operações de * em P e Q, que são variáveis do tipo ponteiro, retornam valores inteiros que compõem uma expressão aritmética. Quando a expressão aritmética é resolvida, seu resultado é um valor inteiro que pode ser inserido em I.

5. Estrutura de Dados Lista

1. Algoritmo para *inserir* um elemento em uma lista duplamente encadeada circular.

```

procedimento inserir (*PR:PRIM)
    inicio
        ATUAL, NOVO, ANTERIOR: NO;
        NOME: caracter[40];
        Aloque (&NOVO);
        imprima("ENTRE COM UM NOME : ");
        leia(NOME);
        se *PR = nulo
            então imprima("lista vazia, inserindo 1º nome");
            *PR ← &NOVO;
            NOVO.NOME ← NOME;
            NOVO.*ANT ← *PR;
            NOVO.*POS ← *PR;
        Senão ATUAL ← *PR;
        enquanto ATUAL.*POS <> nulo faça
            ANTERIOR ← * (ATUAL.*ANT);
            ATUAL ← * (ATUAL.*POS);
        Fim enquanto;
        imprima ("valor será inserido no final da lista");
    
```

Estrutura de Dados

```
NOVO.NOME ← NOME;
ATUAL.*POS ← &NOVO;
NOVO.*ANT ← &ATUAL;
NOVO.*POS ← *PR;
Fimse;
Fim; //procedimento inserir
```

2. Algoritmo para *remover* um elemento de uma lista duplamente encadeada circular.

```
procedimento remover (*PR:PRIM)
    início
        ATUAL, ANTERIOR, AUX: NO;
        NOME: caracter[40];
        Imprima ("digite nome a ser removido");
        Leia (NOME);
        se *PR = nulo
            então imprima("lista vazia, não existe nome para
            remover");
            remover;
        Senão ATUAL ← *PR;
            enquanto ATUAL.*POS <> *PR E ATUAL.NOME <> NOME
            faça
                ANTERIOR ← ATUAL;
                ATUAL ← * (ATUAL.*POS);
            Fim enquanto;
            Se ATUAL.*POS = *PR E ATUAL.NOME <> NOME
                Então imprima ("Nome não encontrado na lista");
                remover;
            Senão se ATUAL.*POS = *PR E ATUAL.NOME = NOME
                Então imprima ("Nome encontrado e removido
                da 1ª posição da lista");
                AUX ← * (ATUAL.*POS);
                AUX.*ANT ← ATUAL.*ANT;
                *PR ← ATUAL.*POS;
                Desaloque (&ATUAL);
            Senão se ATUAL.*POS = *PR
                Então imprima ("nome encontrado e
                removido da última posição da lista");
                ANTERIOR.*POS <- *PR;
                Desaloque (&ATUAL);
            Senão imprima ("nome encontrado e
            removido do meio da lista");
                ANTERIOR.*POS ← ATUAL.*POS;
```

```

        AUX ← *ATUAL.POS;
        AUX.*ANT ← &ANTERIOR;
        Desaloque (&ATUAL);
        Fimse;
        Fimse;
        Fimse;
        Fimse;
        Fim; //procedimento remover

```

3. Algoritmo para *buscar* um elemento em uma lista duplamente encadeada circular.

```

procedimento busca (*PR:PRIM, CHAVE:caracter[40])
    inicio
        ATUAL: NO;
        se *PR = nulo
            então imprima("lista vazia!");
            busca;
        senão ATUAL ← *PR;
            enquanto ATUAL.*PROX <> nulo E ATUAL.NOME <>
CHAVE faça
            ATUAL ← * (ATUAL.*PROX);
            Fim enquanto;
            Se ATUAL.*PROX = nulo
                Então imprima ("Nome não encontrado na lista");
                Busca;
            Senão imprima ("Nome encontrado no endereço ",
&ATUAL, "da lista");
            Fimse;
            Fimse;
        Fim; //procedimento busca

```

4. Algoritmo para *imprimir* uma lista duplamente encadeada circular, nas ordens crescente e decrescente.

```

procedimento imprimir_crescente (*PR:PRIM)
    inicio
        ATUAL: NO;
        se *PR = nulo
            então imprima("lista vazia!");
            imprimir_crescente;
        senão ATUAL ← *PR;
            enquanto ATUAL.*POS <> nulo faça
                imprima (ATUAL.NOME);

```

Estrutura de Dados

```
        ATUAL ← * (ATUAL.*POS) ;
        Fimenquanto;
        Fimse;
Fim; //procedimento imprimir_crescente

procedimento imprimir_decrecente (*PR:PRIM)
    início
    ATUAL: NO;
    se *PR = nulo
        então imprima("lista vazia!");
            imprmir_descrescete;
        senão ATUAL ← *PR;
            enquanto ATUAL.*ANT <> nulo faça
                imprima (ATUAL.NOME);
                ATUAL ← * (ATUAL.*ANT);
            Fimenquanto;
        Fimse;
Fim; //procedimento imprimir_decrecente
```

6. Recursividade

1. Encontrando as condições de parada e recursão:

$N^0 = 1$, então quando o expoente chegar em 0 a função retorna 1 e indica condição de parada.

N^E , quando $E > 0$ multiplicar N por E vezes, indica condição de chamada recursiva.

```
Função POT (N, E) : inteiro
    N, E: inteiro;
    Inicio
        Se E = 0
            Então POT ← 1;
            Senão POT ← N * POT (N, (E - 1));
        Fimse;
    fim.
```

2. Encontrando as condições de parada e recursão:

$X * 0 = 0$, então quando $Y = 0$ a função retorna 0 e indica condição de parada.

$X * 1 = X$, então quando $Y = 1$ a função retorna X e indica condição de parada.

$X * Y$, quando $Y > 1$ somar X por Y vezes, indica condição de chamada recursiva.

```
Função MULT (X, Y) : inteiro
    X, Y: inteiro;
    Inicio
        Se Y = 0
            Então MULT ← 0;
        Senão se Y = 1
            Então MULT ← X;
        Senão MULT ← X + MULT (X, (Y - 1));
        Fimse;
    Fimse;
fim.
```

3. Para resolver o exercício 3, é preciso trocar as chamadas recursivas dos exercícios 1 e 2, respectivamente, por estruturas de repetição (para, enquanto ou repita) respeitando-se as condições de parada encontradas.

a) Potência

```
Função POT (N, E) : inteiro
    N, E: inteiro;
    Inicio
        I : inteiro;
        POT ← 1;
        Para I de 1 até E faça
            POT ← POT * N;
        Fimpara;
Fim.
```

b) Produto

```
Função MULT (X, Y) : inteiro
    X, Y: inteiro;
    Inicio
        I: inteiro;
        MULT ← 0;
        para I de 1 até Y faça
            MULT ← MULT + X;
```

Estrutura de Dados

```
Fim para;  
fim.
```

4.

- a) Indique o resultado do algoritmo com A valendo 5, 1 e 0;

Teste de mesa com A valendo 5, 1 e 0, retorna os valores 15, 1 e 0, respectivamente. Acompanhe no algoritmo apresentado no enunciado o valor das chamadas e retorno da função X nas tabelas a seguir:

Chamada recursiva	Retorno da chamada
X (5)	15
5 + X(4)	5 + 4 + 3 + 2 + 1 + 0
4 + X(3)	4 + 3 + 2 + 1 + 0
3 + X(2)	3 + 2 + 1 + 0
2 + X(1)	2 + 1 + 0
1 + X(0)	1 + 0
0	0

Chamada recursiva	Retorno da chamada
X (1)	1
1 + X(0)	1 + 0
0	0

Chamada recursiva	Retorno da chamada
X (0)	0
0	0

- b) Substituindo as chamadas recursivas da função X do enunciado pelo comando de repetição Para utilizando a variável de controle I:

```
Função X (A) : inteiro;  
A: inteiro;  
Inicio  
    I : inteiro;  
    X ← 0;
```

```

Para I de 1 até A faça
    X ← X + I;
Fimpara;
Fim.

```

Teste de mesa com o valor de A valendo 5:

X	A	I
0	5	1
1		2
3		3
6		4
10		5
15	retorno de X	6 finaliza o Para

Teste de mesa com o valor 1 para A:

X	A	I
0	1	1
1	retorno de X	2 finaliza o Para

Teste de mesa com o valor 0 para A:

X	A	I
0	0	1
0 retorno de X		Para não é executado pois $1 > 0$

7. Árvores

- É mais comum percorrer uma árvore no percurso em-ordem. Ele permite que se extraia dados de uma árvore de forma útil, de modo

que possa ser exibido ou copiado para outra estrutura de dados. Os outros percursos são menos usados.

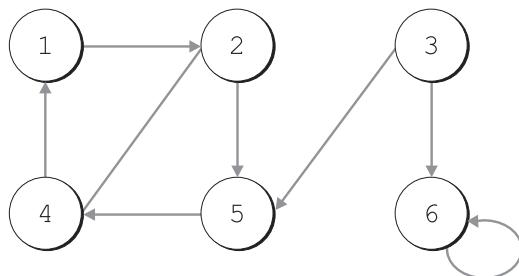
2. As árvores são a estrutura de dados mais útil. Elas têm comparativamente, busca, inserção e remoção rápidas, o que não é o caso com as estruturas de dados mais simples, como vetores e listas encadeadas. Para armazenar quantidades grandes de dados, uma árvore é a primeira estrutura que precisamos considerar.
3. Uma árvore binária de busca tem como característica que seus elementos estejam ordenados na forma de: menores valores na subárvore esquerda e maiores ou iguais valores na subárvore direita. Desta forma, sempre que um novo elemento for inserido na árvore, esta característica precisa ser obedecida. Então, inserindo-se o nó na posição correta da árvore, ela sempre estará organizada. Porém, ao remover um nó da árvore, a reorganização dos nós que ficaram é necessária.

4. $3 + 5 * 8$

$(3 + 5) * 8$

8. Grafos

1.



Nós: 2 e 3

Arestas: (1,2) (3,5)

Percorso: (1,4,2,5) (3,5,4,2)

Ciclo: (2,5,4,2)

Laço: (6,6)

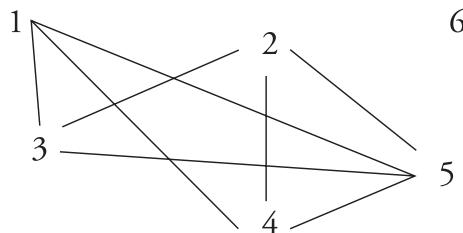
2.

```

Algoritmo // grau do vértice
inicio
    MAX ← 10;
    TIPO M = MATRIZ [1:MAX,1:MAX] inteiro;
    GRAFO : M;
    I, J, GRAU :inteiro;
    Leia (GRAFO);
    Para I de 1 até MAX faça
        GRAU ← 0;
        Para J de 1 até MAX faça
            Se GRAFO [I,J] = 1
                Então GRAU ← GRAU + 1;
            Fimse;
        Fimpara;
        Imprima (" o vértice ", I, "tem grau ", GRAU);
    Fimpara;
Fim.

```

3.



4.

Cabeçalho lista de adjacência

<table border="1"><tr><td>1</td><td></td></tr><tr><td></td><td></td></tr></table>	1				→	<table border="1"><tr><td>2</td><td></td></tr><tr><td></td><td></td></tr></table>	2				→	<table border="1"><tr><td>3</td><td></td></tr><tr><td></td><td></td></tr></table>	3				→	<table border="1"><tr><td>4</td><td>/</td></tr><tr><td></td><td></td></tr></table>	4	/		
1																						
2																						
3																						
4	/																					
<table border="1"><tr><td>2</td><td></td></tr><tr><td></td><td></td></tr></table>	2				→	<table border="1"><tr><td>1</td><td></td></tr><tr><td></td><td></td></tr></table>	1					<table border="1"><tr><td>4</td><td>/</td></tr><tr><td></td><td></td></tr></table>	4	/			→	<table border="1"><tr><td>5</td><td>/</td></tr><tr><td></td><td></td></tr></table>	5	/		
2																						
1																						
4	/																					
5	/																					
<table border="1"><tr><td>3</td><td></td></tr><tr><td></td><td></td></tr></table>	3				→	<table border="1"><tr><td>1</td><td></td></tr><tr><td></td><td></td></tr></table>	1				→	<table border="1"><tr><td>4</td><td></td></tr><tr><td></td><td></td></tr></table>	4				→	<table border="1"><tr><td>5</td><td>/</td></tr><tr><td></td><td></td></tr></table>	5	/		
3																						
1																						
4																						
5	/																					
<table border="1"><tr><td>4</td><td></td></tr><tr><td></td><td></td></tr></table>	4				→	<table border="1"><tr><td>1</td><td></td></tr><tr><td></td><td></td></tr></table>	1					<table border="1"><tr><td>2</td><td></td></tr><tr><td></td><td></td></tr></table>	2				→	<table border="1"><tr><td>3</td><td>/</td></tr><tr><td></td><td></td></tr></table>	3	/		
4																						
1																						
2																						
3	/																					
<table border="1"><tr><td>5</td><td></td></tr><tr><td></td><td></td></tr></table>	5				→	<table border="1"><tr><td>3</td><td></td></tr><tr><td></td><td></td></tr></table>	3					<table border="1"><tr><td>4</td><td>/</td></tr><tr><td></td><td></td></tr></table>	4	/			→	<table border="1"><tr><td>5</td><td>/</td></tr><tr><td></td><td></td></tr></table>	5	/		
5																						
3																						
4	/																					
5	/																					
<table border="1"><tr><td></td><td>/</td></tr><tr><td></td><td></td></tr></table>		/																				
	/																					

9. Métodos de Ordenação e Pesquisa

1. Trocando os comandos enquanto por para no método bolha:

```
inicio //método bolha
    tipo VET = VETOR [ 1:5 ] inteiro;
    V: VET;
    I, AUX, LIM : inteiro;
    leia (V);
    para LIM de 5 até 2 passo -1 faça
        para I de 2 até LIM faça
            se V[I] < V[I-1]
                então AUX ← V[I-1];
                V[I-1] ← V [I];
                V[I] ← AUX;
            fimse;
        fimpara;
    fimpara;
    imprima (V);
fim.
```

2. Trocando os comandos enquanto por para no método seleção simples:

```
inicio // seleção simples
    tipo VET = Vetor [1:5] inteiro;
    V:VET;
    LIM, INDCOR, I, J: inteiro;
    CORR: inteiro;
    LIM ← 5;
    para I de 1 até LIM faça
        CORR ← V[I];
        INDCOR ← I;
        para J de I+1 até LIM faça
            se V[J] < CORR
                então CORR ← V[J];
                INDCOR ← J;
            fimse;
        fimpara;
        AUX ← V[I];
        V[I] ← V[INDCOR];
        V[INDCOR] ← AUX;
    fimpara;
fim.
```

3. Seguem as comparações na tabela:

Método de ordenação	Resultado (utilizando o VisuAlg)	Comentários																																																																	
Bolha	<table border="1"> <tr><td>S</td><td>A</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>D</td><td>U</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>D</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>S</td><td>D</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>S</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>E</td><td>S</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>E</td><td>S</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>E</td><td>S</td><td>U</td></tr> <tr><td colspan="5">Número de trocas: 5</td></tr> </table>	S	A	U	D	E	A	S	U	D	E	A	S	U	D	E	A	S	D	U	E	A	S	D	E	U	A	S	D	E	U	A	D	S	E	U	A	D	E	S	U	A	D	E	S	U	A	D	E	S	U	Número de trocas: 5					Realiza 5 trocas, porém passa o vetor 10 vezes, o que torna o processamento lento e gera desperdício.										
S	A	U	D	E																																																															
A	S	U	D	E																																																															
A	S	U	D	E																																																															
A	S	D	U	E																																																															
A	S	D	E	U																																																															
A	S	D	E	U																																																															
A	D	S	E	U																																																															
A	D	E	S	U																																																															
A	D	E	S	U																																																															
A	D	E	S	U																																																															
Número de trocas: 5																																																																			
Seleção Simples	<table border="1"> <tr><td>S</td><td>A</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>D</td><td>U</td><td>S</td><td>E</td></tr> <tr><td>A</td><td>D</td><td>E</td><td>S</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>E</td><td>S</td><td>U</td></tr> <tr><td colspan="5">Número de trocas: 4</td></tr> </table>	S	A	U	D	E	A	S	U	D	E	A	D	U	S	E	A	D	E	S	U	A	D	E	S	U	Número de trocas: 4					Realiza 4 trocas e passa o vetor 4 vezes. É eficiente.																																			
S	A	U	D	E																																																															
A	S	U	D	E																																																															
A	D	U	S	E																																																															
A	D	E	S	U																																																															
A	D	E	S	U																																																															
Número de trocas: 4																																																																			
Inserção	<table border="1"> <tr><td>S</td><td>A</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>S</td><td>A</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>D</td><td>S</td><td>U</td><td>E</td></tr> <tr><td>A</td><td>D</td><td>E</td><td>S</td><td>U</td></tr> <tr><td colspan="5">Número de trocas: 5</td></tr> </table>	S	A	U	D	E	S	A	U	D	E	A	S	U	D	E	A	S	U	D	E	A	D	S	U	E	A	D	E	S	U	Número de trocas: 5					Realiza 5 trocas e passa o vetor 5 vezes.																														
S	A	U	D	E																																																															
S	A	U	D	E																																																															
A	S	U	D	E																																																															
A	S	U	D	E																																																															
A	D	S	U	E																																																															
A	D	E	S	U																																																															
Número de trocas: 5																																																																			
Oscilante	<table border="1"> <tr><td>S</td><td>A</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>U</td><td>D</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>D</td><td>U</td><td>E</td></tr> <tr><td>A</td><td>S</td><td>D</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>S</td><td>D</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>S</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>S</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>S</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>S</td><td>E</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>E</td><td>S</td><td>U</td></tr> <tr><td>A</td><td>D</td><td>E</td><td>S</td><td>U</td></tr> <tr><td colspan="5">Número de trocas: 5</td></tr> </table>	S	A	U	D	E	A	S	U	D	E	A	S	U	D	E	A	S	D	U	E	A	S	D	E	U	A	S	D	E	U	A	D	S	E	U	A	D	S	E	U	A	D	S	E	U	A	D	S	E	U	A	D	E	S	U	A	D	E	S	U	Número de trocas: 5					Realiza 5 trocas porém passa o vetor 14 vezes. Percebe-se que em alguns casos o Bolha pode ser mais eficiente do que o Oscilante. Depende muito do conteúdo do vetor.
S	A	U	D	E																																																															
A	S	U	D	E																																																															
A	S	U	D	E																																																															
A	S	D	U	E																																																															
A	S	D	E	U																																																															
A	S	D	E	U																																																															
A	D	S	E	U																																																															
A	D	S	E	U																																																															
A	D	S	E	U																																																															
A	D	S	E	U																																																															
A	D	E	S	U																																																															
A	D	E	S	U																																																															
Número de trocas: 5																																																																			

4. Seguem os resultados na tabela:

Pesquisa	Resultado (utilizando o Visualg)
Sequencial com vetor não ordenado	<pre> S A U D E Chave: D chave encontrada na posição 4 </pre>
Sequencial vetor ordenado	<pre> A D E S U Chave: Z chave não encontrada no vetor </pre>
Binária vetor obrigatoriamente ordenado	<pre> A D E S U Chave: D chave encontrada na posição 2 </pre> <pre> A D E S U Chave: Z chave não encontrada no vetor </pre>

10. Complexidade de Algoritmos

1.

Método	Complexidade
Bolha	$O(n^2)$
Inserção direta	$O(n^2)$
ShellSort	$O(n^{4/3})$
QuickSort	$O(n \log(n))$ de tempo $O(\log(n))$ de espaço

Método	Complexidade
MergeSort	$O(n \log(n))$ de tempo
	$O(n + \log(n))$ de espaço
HeapSort	$O(n \log(n))$

2. Quando analisamos o pior caso, estamos verificando a análise da complexidade do algoritmo com o maior número de operações usadas para qualquer entrada de tamanho n . Assim, isso garante que o algoritmo termine no tempo. Além disso, a ordem de crescimento da complexidade de pior caso pode ser usada para comparar a eficiência entre dois algoritmos.
3. Sim, o algoritmo B poderia ser utilizado pois, apesar da complexidade do algoritmo B ser exponencial, quando o valor de n é pequeno, este algoritmo produz um tempo de complexidade menor do que o algoritmo A. Isto pode ocorrer com valores de n iguais a 2, 10 e 20, por exemplo.
4. Analisando as complexidades de maneira simplificada e considerando que n é o tamanho da entrada, se substituirmos o valor de n por um número inteiro, como 10, é possível ter uma ideia da magnitude de cada uma das funções de complexidade.

Empresa	Complexidade	Resultado com $n=10$
Alfa	$O(n^{20})$	100.000.000.000.000.000.000.000.000.000
Sigma	$O(1)$	1
Delta	$O(n \log n)$	$10 * 1 = 10$
Gama	$O(n!)$	3.628.800
Omega	$O(2^n)$	1024

Podemos então ordenar as respostas dos valores de complexidade por grandeza, como: $1 < 10 < 1024 < 3.628.800 < 100.000.000.000.000.000.000$, então $1 < 10 \log 10 < 2^{10} < 10! < 10^{20}$. Podemos concluir que a empresa que ficou em segundo lugar na lista do setor de TI com complexidade $O(n \log n)$ foi a empresa Delta.

Referências

CORMEN, T. H. et al. **Algoritmos**: teoria e prática. 3. ed. Editora Campus, 2012.

FORBELLONE, A. L. V.; EBERSPACHER, H. F. **Lógica de programação**: a construção de algoritmos e estruturas de dados. 3. ed. Makron Books, 2005.

INFOESCOLA. **Máximo divisor comum (MDC)**. Disponível em: <<http://www.infoescola.com/matematica/maximo-divisor-comum-mdc/>>. Acesso em: 20 nov. 2016.

_____. **Sequência de Fibonacci**. Disponível em: <<http://www.infoescola.com/matematica/sequencia-de-fibonacci/>>. Acesso em: 20 nov. 2016.

KANTEK, P. DE BASSI, P. R. **Algoritmos em programação estruturada**, material de aula, 2013.

LAFORE, R. **Estrutura de dados e algoritmos**. Editora Campus, 1999.

LAUREANO, M. A. P. **Estrutura de Dados com Algoritmos e C**. Brasport, 2008.

MONTEBELLO JUNIOR, M. A. **Grafos definições preliminares**. 2014. Disponível em: <<http://slideplayer.com.br/slide/67252/>>. Acesso em: 28 nov. 2016.

ORTIZ, A. **Eficiência e eficácia**. Disponível em: <http://www.infoescola.com/administracao/_eficiencia-e-eficacia/>. Acesso em: 20 dez. 2016.

SORTING algorithms animations. Disponível em: <<https://www.toptal.com/developers/sorting-algorithms/>>. Acesso em: 25 nov. 2016.

TENENBAUM A. M.; LANGSAM, Y.; AUGESTEIN, M. J. **Estruturas de dados usando C**. Makron Books, 1995.

VISUALG. Disponível em: <<http://www.apoioinformatica.inf.br/produtos/visualg>>. Acesso em: 25 nov. 2016.

ZIVIANI, N. **Projeto de Algoritmos com Implementações em Pascal e C**. 3. ed. Cengage Learning, 2010.

É sempre bom ressaltar que o computador é uma máquina que processa dados. Como o próprio nome da disciplina diz, a nossa maior preocupação é em como estruturar os dados no computador. Na verdade, existe um local específico em que estamos interessados: a memória do computador.

Nesse contexto, podemos sintetizar a estrutura de dados como a forma de organizarmos nossos dados na memória do computador. É uma ideia simples, afinal, devemos ser organizados em tudo, inclusive em como utilizar a memória do computador.

A organização se reflete na economia de recursos computacionais e na diminuição da complexidade de resolução de determinados problemas. Por isso, existem algumas estruturas clássicas, amplamente utilizadas e que serão estudadas neste livro, como pilhas, filas, listas, árvores e grafos.

Além disto, para escrevermos algoritmos eficientes, é importante conhecer alguns recursos de programação, como a alocação de memória (uso dos ponteiros) e a recursividade. Para validarmos a eficiência de nossos algoritmos, vamos conhecer sobre complexidade de algoritmos.

Você agora tem o computador e a teoria nas mãos. Aproveite!



EDITORIA
FAEL

ISBN 978-85-60531-95-0



9 788560 531950