

ALGORITMOS E PROGRAMAÇÃO II

Dados Internacionais de Catalogação na Publicação (CIP)
(Jeane Passos de Souza - CRB 8ª/6189)

Masiero, Andrey Araujo

Algoritmos e programação II / Andrey Araujo Masiero. – São Paulo :
Editora Senac São Paulo, 2020. (Série Universitária)

Bibliografia.

e-ISBN 978-65-5536-123-0 (ePub/2020)

e-ISBN 978-65-5536-124-7 (PDF/2020)

1. Desenvolvimento de sistemas 2. Linguagem de programação
3. Algoritmos : Programação 4. Programação recursiva I. Título.
II. Série

20-1131t

CDD – 005.13

003

BISAC COM051300

COM051230

Índice para catálogo sistemático

1. Linguagem de programação : Algoritmos 005.13

2. Desenvolvimento de sistemas 003

ALGORITMOS E PROGRAMAÇÃO II

Andrey Araujo Masiero





Administração Regional do Senac no Estado de São Paulo

Presidente do Conselho Regional

Abram Szajman

Diretor do Departamento Regional

Luiz Francisco de A. Salgado

Superintendente Universitário e de Desenvolvimento

Luiz Carlos Dourado

Editora Senac São Paulo

Conselho Editorial

Luiz Francisco de A. Salgado

Luiz Carlos Dourado

Darcio Sayad Maia

Lucila Mara Sbrana Sciotti

Jeane Passos de Souza

Gerente/Publisher

Jeane Passos de Souza (jpassos@sp.senac.br)

Coordenação Editorial/Prospecção

Luís Américo Tousi Botelho (luis.tbotelho@sp.senac.br)

Márcia Cavalheiro Rodrigues de Almeida (mcavalhe@sp.senac.br)

Administrativo

João Almeida Santos (joao.santos@sp.senac.br)

Comercial

Marcos Telmo da Costa (mtcosta@sp.senac.br)

Acompanhamento Pedagógico

Ariádney Carolina Brasileiro

Designer Educacional

Sueli Brianezi Carvalho

Revisão Técnica

Gustavo Moreira Calixto

Coordenação de Preparação e Revisão de Texto

Luiza Elena Luchini

Preparação de Texto

Ana Luiza Candido

Revisão de Texto

Ana Luiza Candido

Projeto Gráfico

Alexandre Lemes da Silva

Emília Corrêa Abreu

Capa

Antonio Carlos De Angelis

Editoração Eletrônica

Michel Iuiti Navarro Moreno

Ilustrações

Michel Iuiti Navarro Moreno

Imagens

iStock Photos

E-pub

Ricardo Diana

Proibida a reprodução sem autorização expressa.

Todos os direitos desta edição reservados à

Editora Senac São Paulo

Rua 24 de Maio, 208 – 3º andar

Centro – CEP 01041-000 – São Paulo – SP

Caixa Postal 1120 – CEP 01032-970 – São Paulo – SP

Tel. (11) 2187-4450 – Fax (11) 2187-4486

E-mail: editora@sp.senac.br

Home page: <http://www.livrariasenac.com.br>

© Editora Senac São Paulo, 2020

Sumário

Capítulo 1 **Métodos de busca e** **classificação de dados em** **memória, 7**

- 1 Busca sequencial, 8
 - 2 Busca binária: a mais rápida entre as buscas, 19
 - 3 Exercícios de fixação, 22
- Considerações finais, 23
- Referências, 23

Capítulo 2 **Técnicas de programação** **recursiva, 25**

- 1 Definição da recursão, 26
 - 2 Traga a matemática à vida, 28
 - 3 Tipos de recursão, 36
 - 4 Iteração ou recursão?
Eis a questão, 42
 - 5 Exercícios de fixação, 43
- Considerações finais, 43
- Referências, 44

Capítulo 3 **Algoritmos de ordenação** **simples, 45**

- 1 Bubble sort, 47
 - 2 Insertion sort, 53
 - 3 Selection sort, 56
 - 4 Exercícios de fixação, 60
- Considerações finais, 60
- Referências, 61

Capítulo 4 **Algoritmos de ordenação** **sofisticados, 63**

- 1 Merge sort, 66
 - 2 Quicksort, 70
 - 3 Exercícios de fixação, 75
- Considerações finais, 75
- Referências, 76

Capítulo 5 **Eficiência de algoritmos, 77**

- 1 Matemática como ferramenta para análise de algoritmos, 79
 - 2 Comparativo das taxas de crescimento, 85
 - 3 Identificando a função do algoritmo, 88
 - 4 Notação Big-O, 91
 - 5 Exercícios de fixação, 93
- Considerações finais, 94
- Referências, 95

Capítulo 6 **Filas, 97**

- 1 Implementando a fila com vetores, 100
 - 2 Andando em círculo, 108
 - 3 Ligando os elementos, 112
 - 4 Exercícios de fixação, 116
- Considerações finais, 117
- Referências, 117

Capítulo 7 **Pilha, 119**

- 1 Implementando a pilha com o uso de vetores, 122
 - 2 Utilizando uma lista ligada para empilhar objetos, 130
 - 3 Exercícios de fixação, 134
- Considerações finais, 135
- Referências, 135

Capítulo 8 **Fila de prioridade e heap, 137**

- 1 Implementando a fila de prioridades com vetores, 140
 - 2 Heap, 149
 - 3 Exercícios de fixação, 159
- Considerações finais, 159
- Referências, 160

Sobre o autor, 163

Métodos de busca e classificação de dados em memória

Uma das operações que mais realizamos no dia a dia é a busca por informações. Desde o momento em que acordamos até a hora de dormir, é difícil passarmos o dia sem fazermos uma busca. Nomes na agenda, cereal no armário, comida na geladeira, lápis no estojo, entre muitas outras. Nas aplicações computacionais isso não é diferente: a busca é a tarefa mais realizada. Sempre é necessário encontrar algum dado na memória ou em qualquer dispositivo de armazenamento.

Para realizar um processo de busca, é necessário um conjunto de dados que geralmente é estruturado com um vetor (WIRTH, 1989). Um vetor de objetos v é definido pela equação a seguir:

$$v: \text{VETOR}[0 \dots n - 1] \text{ de } \text{OBJETOS}$$

Em que v é um vetor que possui n objetos, classificados com índices de 0 até $n - 1$.

Essa definição é dada considerando que esse é um conjunto de dados de tamanho fixo, para facilitar a compreensão de todo o processo. Todo objeto existente no vetor v é inserido junto com uma chave de busca c (WIRTH, 1989). Um objeto o pode conter quantos atributos forem necessários para definir o problema a ser tratado. Aconselha-se que sua chave de busca seja um valor numérico, inteiro e positivo. Assim, as comparações para uma busca são mais eficientes devido ao tipo de dado.

Graficamente, é natural definir o vetor como uma barra retangular dividida em pequenos quadrados, os quais representam os objetos armazenados no vetor. A figura 1 apresenta a ilustração do vetor descrito, no qual os objetos estão armazenados a partir da posição de chave 0 aumentando até o valor final de chave $n - 1$. É importante salientar que n é o número de objetos armazenados no vetor.

Figura 1 – Ilustração de um vetor com 10 posições

0(0)	0(1)	0(2)	0(3)	0(n - 2)	0(n - 1)
------	------	------	------	-----	-----	-----	-----	----------	----------

Existem dois principais métodos para realizar buscas em um vetor de dados. A busca sequencial e a busca binária são as duas formas mais utilizadas. Além delas, algumas pequenas variações desses algoritmos serão apresentadas ao longo do capítulo. As implementações aqui expostas utilizaram a linguagem de programação Java, a qual pode ser facilmente traduzida para outras linguagens como C, C++, JavaScript, Python, entre outras.

1 Busca sequencial

Entre os métodos de busca, a busca sequencial, ou linear, é a que tem a implementação mais intuitiva. Como o nome já descreve, a busca

é realizada em cada uma das posições existentes no conjunto de dados, estruturados com vetor, até que se encontre a posição desejada do valor procurado. As condições de parada para esse algoritmo são determinadas por dois critérios básicos (WIRTH, 1989):

- o elemento procurado é encontrado;
- o vetor é completamente analisado, ou seja, o elemento procurado não é encontrado.

Geralmente, dois tipos abstratos de dados (TAD) são empregados nesse tipo de algoritmo. Ambos os tipos podem ser denominados como uma tabela de dados organizada. O primeiro TAD é o vetor no qual o uso de memória de armazenamento é realizado de forma estática (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Isso é o espaço utilizado por um vetor na memória, determinado antes da compilação do sistema para uso.

A segunda opção é utilizar o TAD lista ligada, que possui um tipo de armazenamento de informação dinâmico (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Este funciona com uma estrutura de ligação de nós (objetos), sendo assim, o consumo de memória é mais efetivo, uma vez que os objetos são criados e conectados de acordo com a necessidade do sistema. Nos exemplos que serão apresentados ao longo deste capítulo, o TAD escolhido será o vetor, pois, por possuir uma sintaxe mais simples e prática do que a da lista ligada, é mais fácil compreendê-lo.

Vamos apresentar um vetor contendo cinco elementos armazenados. Em seguida, será realizada a busca por um elemento existente e por outro não existente no vetor. Cada posição do vetor possui um índice que é a chave de acesso às informações armazenadas. Veja a ilustração gráfica do vetor e suas chaves de acesso na figura 2: existem cinco quadrados para armazenar as informações do vetor, que no exemplo em questão são números. Embaixo de cada quadrado existente há um número chamado de índice, o qual determina o valor de acesso à informação.

Figura 2 – Representação gráfica de um vetor com cinco elementos

35	15	29	83	10
0	1	2	3	4

Para realizar o processo de percorrer um vetor, utiliza-se uma estrutura de laço de repetição. Esse tipo de estrutura é encontrado em todas as linguagens de programação, mesmo que com diferentes nomenclaturas e padrões de declarações. Em linguagens derivadas da linguagem C, é comum encontrar três tipos de laços de repetição. São eles: *for*, *while* e *do-while*. É possível obter o mesmo resultado com qualquer um desses laços de repetição. Por uma questão didática, ao longo deste capítulo serão utilizados apenas os laços de repetição *while* e *for*.

Para implementar a busca sequencial, primeiro é necessário criar o vetor em memória. O código será implementado na linguagem Java e utilizará como exemplo o mesmo vetor demonstrado na figura 2. O vetor é declarado a seguir:

```
int vetor[] = {35, 15, 29, 83, 10};
```

Agora, vamos declarar o valor a ser procurado no vetor:

```
int valorProcurado = 83;
```

O próximo passo é construir o método para a busca sequencial. O método recebe como parâmetro um vetor do mesmo tipo do declarado e o valor procurado, que, no exemplo, é um número inteiro. O método deve retornar um valor inteiro correspondente à posição que o valor procurado se encontra no vetor. Se o valor procurado não existir no vetor, o

valor retornado deve ser -1 , isso porque não existe um índice ou posição negativa em um vetor.

```
int buscaSequencial(int[] vetor, int valorProcurado) {  
    for (int i = 0; i < vetor.length; i++) {  
        if (vetor[i] == valorProcurado) {  
            return i;  
        }  
    }  
    return -1;  
}
```

A instrução *for* é utilizada por facilitar a implementação da busca pelo valor. Nela a variável *i* é declarada e inicializada com o valor 0, que representa a primeira posição do vetor. O critério de parada é determinado pela condição *i* menor que o tamanho do vetor. Enquanto essa condição for verdadeira, será executado o bloco de código determinado entre chaves. A cada iteração o valor de *i* é incrementado em 1, fazendo com que o vetor seja percorrido posição a posição.

Quando o valor armazenado na posição *i* do vetor for igual ao valor procurado, condição da instrução *if*, o valor atual de *i* é retornado pelo método *buscaSequencial*. Se em nenhum momento a condição for verdadeira, a busca do dado no vetor será finalizada e, com a finalização do *for*, o valor -1 é retornado para o programa. O passo a passo da procura pelo valor 83 é apresentado através da forma gráfica, iniciando pela figura 3, na qual são apresentados o vetor de busca, o valor procurado, o valor de *i* e também o resultado da comparação existente dentro da condição do *if* no algoritmo:

Figura 3 – Passo 1 no processo da busca sequencial

35	15	29	83	10
0	1	2	3	4

Comparação:
35 == 83 -- Falso

83	Valor Procurado
0	<i>i</i>

Figura 4 – Passo 2 no processo da busca sequencial

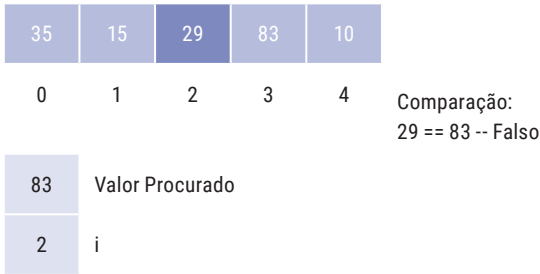
35	15	29	83	10
0	1	2	3	4

Comparação:
15 == 83 -- Falso

83	Valor Procurado
1	<i>i</i>

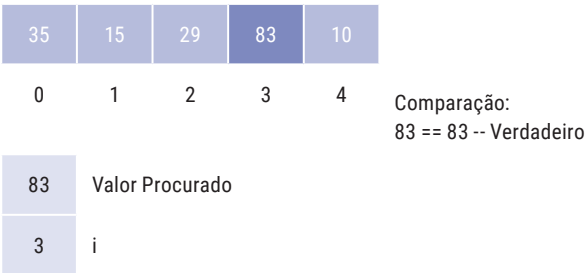
Agora, na figura 4, na qual são apresentados o vetor de busca, o valor procurado, o valor de *i* e também o resultado da comparação existente dentro da condição do *if* no algoritmo, a posição de número 1 é acionada para comparação (quadrado mais escuro). O espaço de memória *i* recebe o valor de 1 (posição atual do vetor no algoritmo). Neste passo, a comparação continua com o resultado falso, portanto uma nova iteração é necessária. A figura 5 apresenta o próximo passo do algoritmo, no qual são apresentados o vetor de busca, o valor procurado, o valor de *i* e também o resultado da comparação existente dentro da condição do *if* no algoritmo:

Figura 5 – Passo 3 no processo da busca sequencial



O valor armazenado nesta posição é 29, que continua diferente de 83, o valor procurado, portanto a condição é falsa. É necessário incrementar mais uma vez a variável *i* para seguir para a próxima posição do vetor. Acompanhe o passo 4 na figura 6, na qual são apresentados o vetor de busca, o valor procurado, o valor de *i* e também o resultado da comparação existente dentro da condição do *if* no algoritmo:

Figura 6 – Passo 4 no processo da busca sequencial



Após incrementar a variável *i* para o valor 3 e, por consequência, acessar a posição 3 do vetor, o algoritmo irá comparar o valor armazenado com o valor procurado. Neste momento, a comparação será verdadeira e o método deve retornar o valor de *i* para o programa que o chamou.

Apesar de ser um método intuitivo e de baixa complexidade de implementação, nos cenários com casos mais extremos de busca, como a não existência do objeto ou o posicionamento dele na última posição do vetor, será necessária uma quantidade *n* de operações, em que *n* é a quantidade de elementos existentes no vetor. Quando esse número

atingir uma quantidade considerada grande, esse algoritmo torna-se ineficiente, devido ao tempo de processamento. Sendo assim, na próxima seção, serão apresentadas algumas técnicas para otimizar o algoritmo de busca sequencial.

1.1 Otimizando a busca sequencial

Otimizar a busca sequencial é uma tarefa almejada em diversos estudos em ciências da computação. Uma forma de realizá-la é determinar a probabilidade de cada elemento ser pesquisado. Com base nessa probabilidade, deve-se realizar uma ordenação, na qual o elemento com maior probabilidade de ser pesquisado no conjunto seja posicionado entre as primeiras posições do vetor (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). A probabilidade do elemento na posição i é definida pela notação $p(i)$, em que:

$$p(0) \geq p(1) \geq p(2) \geq \dots \geq p(n - 1)$$

Sendo essa a definição das probabilidades de todos os objetos contidos no vetor, a soma de todas as probabilidades deve ser igual a 1, conforme a equação a seguir:

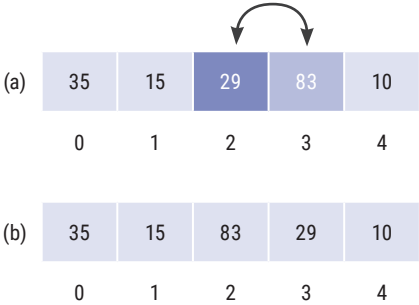
$$\sum_{i=0}^{n-1} p(i) = 1$$

Para armazenar o valor da probabilidade de cada elemento armazenado, é recomendado utilizar um atributo extra ou que seja calculado a partir de alguma regra estabelecida de acordo com um conhecimento prévio sobre os dados armazenados (TENENBAUM; LANGSAM; AUGENSTEIN, 2004).

Entretanto, estabelecer uma regra de probabilidade para operações cotidianas não é uma tarefa simples e pode onerar mais o algoritmo em

vez de otimizá-lo. Para contornar a situação, pode-se utilizar uma regra mais simples que o cálculo de probabilidade. Quando um elemento for encontrado durante a busca, faça a troca dele com o elemento na posição de menor índice (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Para ilustrar, voltemos ao passo 4 da busca sequencial apresentado na figura 6. O valor 83 foi encontrado durante a realização da busca sequencial. Imediatamente, ele trocaria de posição com o valor anterior, no caso 29, se aproximando mais do início do vetor. O processo de troca é apresentado na figura 7, a seguir.

Figura 7 – Processo de priorização dos valores buscados no vetor



A ideia apresentada com o processo de priorização do valor encontrado na busca pode diminuir o tempo de busca de um elemento específico. Contudo, é um procedimento que pode prejudicar os elementos menos buscados (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Se pensarmos em um cenário mais extremo, buscar o elemento que é acessado esporadicamente levará ao mesmo número de comparações que a busca sequencial original, ou seja, n elementos.

Trabalhar com o vetor sem uma ordenação prévia, aparentemente, não torna o algoritmo de busca significativamente mais eficiente que o sequencial tradicional. Algoritmos mais eficientes na operação de busca estabelecem a premissa de que o vetor precisa estar ordenado. Por exemplo, a busca sequencial tradicional já possui um pequeno ganho no caso de um vetor ordenado. No critério de parada da instrução *for*,

é possível adicionar uma condição para verificar se o elemento atual é menor que o procurado. Quando essa condição for falsa, a busca pode terminar, não sendo necessário percorrer o vetor completo. Veja o algoritmo modificado para atender a esse critério:

```
int buscaSequencial(int[] vetor, int valorProcurado) {
    for (int i = 0; i < vetor.length
        && vetor[i - 1] < vetor[i]; i++) {
        if (vetor[i] == valorProcurado) {
            return i;
        }
    }
    return -1;
}
```

Essa alteração não elimina todos os casos de percorrer o vetor completo quando os valores buscados não se encontram no vetor. Porém, é possível eliminar processamentos exaustivos já sabendo o resultado final (ASCENCIO; ARAÚJO, 2010). Pode parecer simples a ideia de ordenar o vetor antes de realizar o processo de busca, mas um vetor ordenado sofre em média $n/2$ operações para encontrar um valor, enquanto o vetor sem ordenação sofre n operações, lembrando que n é a quantidade de elementos armazenados no vetor.



IMPORTANTE

Dado um vetor que possui n elementos, a quantidade média de operações para encontrar um valor é de:

- n comparações para um vetor sem ordenação; e
- $n/2$ comparações para um vetor ordenado.

Um algoritmo que otimiza a busca sequencial tradicional é o da busca sequencial indexada, que será apresentado na seção seguinte.

1.2 Busca sequencial indexada

O algoritmo de busca sequencial indexada utiliza uma tabela auxiliar para otimizar o processo de busca. Essa tabela geralmente é chamada de índice. A tabela índice armazena a posição onde um determinado elemento está contido na tabela geral. Dessa maneira, é possível reduzir o espaço da busca sequencial, tornando-a mais eficiente em sua velocidade (TENENBAUM; LANGSAM; AUGENSTEIN, 2004).



IMPORTANTE

Na busca sequencial indexada, o uso do TAD lista ligada não é recomendado em hipótese alguma. Seu uso pode gerar uma sobrecarga nos ponteiros de memórias e isso provoca instabilidades no sistema operacional.

Na figura 8, é ilustrado um exemplo de busca sequencial indexada, com um trecho da tabela índice e outro da tabela geral.

Figura 8 – Ilustração das tabelas índice e geral utilizadas na busca sequencial indexada

Tabela índice		Tabela geral	
Elemento	Índice	Índice	Elemento
31	6	0	5
62	14	1	8
		2	12
		3	18
		4	23
		5	27
		6	31
		7	35
		8	38
		9	42
		10	46
		11	49
		12	51
		13	55
		14	62

Imagine que o valor procurado no vetor seja 12. O algoritmo deve ir primeiro à tabela índice e verificar se 12 é menor ou igual ao valor da chave de busca armazenada. No exemplo da figura 8, logo na primeira comparação a condição é verdadeira, uma vez que a comparação realizada é “12 é menor ou igual a 31”. Nesse momento, é estabelecido o intervalo de busca na tabela geral, no qual o índice referente ao valor 31 delimitará o último elemento de comparação do vetor.

Parece que não fará diferença nenhuma, afinal, o exemplo ilustrado possui apenas 15 elementos e foi reduzido para 7. Mas se o cenário ultrapassar 1 milhão de elementos, será que o ganho pode ser considerado, justificando a implementação do algoritmo de busca sequencial indexada? Nessa situação, a busca indexada é bem interessante para diminuir o tempo e o espaço de procura no processo. Contudo, vale uma ressalva importante de que a busca sequencial indexada é mais rápida, mas consome mais memória de armazenamento. Isso ocorre pois são criadas novas tabelas menores para o processo anterior à busca na tabela geral. Algoritmos sempre devem ser balanceados entre memória de processamento e memória de armazenamento e nem sempre é possível otimizar os dois. Uma das memórias deve ser sacrificada em benefício da outra.

Durante a implementação de uma tabela índice o uso do vetor não é uma boa escolha, já que a chave de busca é o mesmo elemento armazenado na tabela geral. Sendo assim, uma melhor escolha de implementação é o uso do dicionário de dados, no qual os tipos da chave e o valor são determinados de acordo com a necessidade do problema. Outro ponto importante é estar ciente de que o uso das tabelas índice não tem limites. Podem ser criadas quantas tabelas índice forem necessárias para diminuir o espaço de busca, atentando-se apenas ao uso da memória de armazenamento para não sobrecarregar o sistema.

Apesar de a busca sequencial indexada otimizar a velocidade na busca por um elemento armazenado, ela ainda não é tão eficiente em relação à constância do tempo gasto na tarefa. Para isso, existe um algoritmo que realiza uma busca mais eficiente. Esse algoritmo é o da busca binária, que será discutido em detalhes no próximo tópico.

2 Busca binária: a mais rápida entre as buscas

A maneira de acelerar um processo de busca é obtendo um conhecimento prévio sobre o dado armazenado no vetor ou qualquer outra estrutura de dados. No entanto, é possível acelerar a busca sem esse conhecimento prévio específico sobre cada tipo de dado. Como apresentado anteriormente, a forma mais simples é manter o vetor totalmente ordenado. Esse tipo de cuidado com o armazenamento auxilia em muitos aspectos. Imagine uma lista de nomes fora da ordem alfabética: ela é praticamente inútil (WIRTH, 1989).

Um vetor ordenado tem a seguinte definição formal:

$$V[n]: 1 \leq k < n : a_{k-1} \leq a_k$$

A partir do vetor ordenado, pode-se aplicar uma técnica chamada de busca binária, que consiste em dividir o vetor de dados em dois subvetores. Os objetos que ficam à esquerda do ponto de divisão devem ser menores que ele, e os que estão à direita, maiores. Quando o algoritmo comparar o objeto no ponto de divisão e este for igual ao elemento procurado, ele retorna sua posição no vetor.

Esse ponto de divisão pode ser determinado por quaisquer posições existentes no vetor, sendo sua escolha aleatória ou arbitrária. Essa decisão específica não influenciará o resultado final da busca (WIRTH, 1989). O importante em cada passo é eliminar o maior número de objetos, atingindo o objetivo de maneira mais rápida. Existe uma solução ótima para esse algoritmo: a escolha do ponto de divisão deve ser baseada no ponto central do conjunto de dados.

A estratégia de cortar o conjunto de dados sempre ao meio faz com que o espaço de busca por um objeto saia de n elementos para $\log n$. É um ganho significativo entre os algoritmos existentes, melhor do que esse resultado apenas uma busca em tempo constante. Sendo assim,

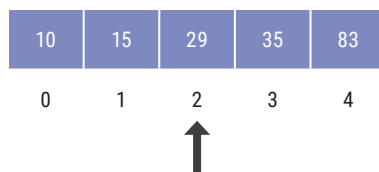
conforme o tamanho dos dados aumenta, o ganho de velocidade da busca binária é computacionalmente expressiva para todo o processo (WIRTH, 1989).

A condição de parada do algoritmo é dada através da seguinte equação:

$$l > F : l \leq m < F$$

Em que l é a posição de início do vetor, portanto menor que m , que representa a posição central do vetor. F é a posição final do vetor, sendo ela maior que m e l . Os passos do algoritmo da busca binária serão apresentados a seguir, iniciando pela figura 9, que representa o passo 1 da busca pelo elemento 83, agora no vetor ordenado.

Figura 9 – Passo 1 do processo do algoritmo da busca binária, onde o centro do vetor é encontrado



O cálculo para encontrar a posição central é repetido novamente. O valor 4 é somado ao 3 e o resultado é dividido por 2. Nesse momento, o resultado obtido é 3,5, porém apenas a parte inteira do valor obtido é considerada, ou seja, 3. O elemento escolhido no passo 2 é apresentado na figura 10, a seguir.

Figura 10 – Passo 2 do processo do algoritmo da busca binária, onde o centro do subvetor é encontrado

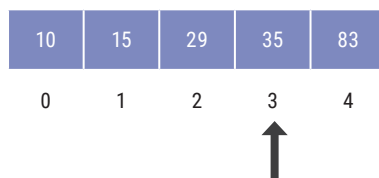



Figura 11 – Passo 3 do processo do algoritmo de busca binária, escolha do elemento no momento em que o vetor dividido possui apenas um elemento

10	15	29	35	83
0	1	2	3	4



Nesse momento, a comparação realizada tem o resultado como verdadeiro. Assim como o algoritmo de busca sequencial, a busca binária retorna agora a posição referente ao elemento procurado, no caso do exemplo, o valor 4. Perceba que, em comparação ao mesmo vetor sem a aplicação da ordenação, o algoritmo fez a busca em uma comparação a menos que a sequencial. Para o vetor ordenado, são exatamente duas comparações a menos. Isso parece pouco no começo, mas com o aumento do volume de dados nota-se um ganho muito grande de desempenho computacional. Veja a seguir a implementação do algoritmo em Java:

```
int buscaBinaria(int[] vetor, int valorProcurado) {
    int inicio = 0;
    int fim = vetor.length - 1;

    while (inicio < fim) {
        int meio = (inicio + fim) / 2;
        if (vetor[meio] > valorProcurado)
            fim = meio - 1;
        else if (vetor[meio] <
valorProcurado)
            inicio = meio + 1;
        else
            return meio;
    }
    return -1;
}
```

É uma implementação relativamente simples, porém o mais importante neste algoritmo refere-se à sua entrada de dados. Todo vetor no processo da busca binária deve ser ordenado. Caso ele não esteja ordenado, o algoritmo poderá finalizar sem realizar a pesquisa de maneira esperada.

3 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. No vetor [2, 5, 7, 9, 10, 8, 92] é possível aplicar a busca binária.
2. Crie um vetor ordenado com aproximadamente mil números inteiros gerados aleatoriamente. Agora, utilize a técnica da busca sequencial indexada para otimizar o processo de busca. Sugestão, utilize uma tabela índice com um intervalo de cem chaves aproximadamente.
3. Com base no exercício 2, crie mais uma tabela índice e veja o comportamento do algoritmo. Qual deles foi o mais eficiente?
4. Qual a quantidade de tabelas índice ideal para obter o melhor desempenho computacional?
5. Qual a condição essencial para que o algoritmo da busca binária seja aplicado no vetor? Por quê?
6. Dados os vetores abaixo, qual o método de busca mais adequado para cada um deles:
 - a. [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
 - b. [30, 28, 25, 22, 19, 16, 12, 8, 6, 1]
 - c. [45, 12, 23, 74, 83, 29, 10, 45]

7. Implemente o algoritmo de busca sequencial, reordenando-o com o elemento encontrado vindo uma posição para a frente.

Considerações finais

Neste capítulo, vimos o quão importante é o processo de busca em atividades cotidianas e também nas principais atividades computacionais. A busca sequencial, apesar de intuitiva, não possui um bom desempenho em conjuntos de dados relativamente grandes. Ela pode ser otimizada quando se cria uma tabela de indexação de alguns elementos existentes no vetor. Isso delimita o espaço de busca, tornando a busca sequencial um pouco mais rápida.

No entanto, esse método só é bem-sucedido com o vetor ordenado. A regra da ordenação é essencial para o funcionamento do algoritmo de busca mais eficiente até aqui: a busca binária. Ela consegue reduzir o espaço de pesquisa de um algoritmo de n comparações para $\log n$. Contudo, quando o vetor possui um tamanho reduzido, a diferença de desempenho entre a busca sequencial e a binária é tão baixa que não faz diferença qual método utilizar. A decisão fica a critério da equipe de desenvolvimento e de acordo com os requisitos do sistema em construção.

Referências

ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela Santos de. **Estruturas de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: MAKRON Books, 2004.

WIRTH, Niklaus. **Algoritmos e estruturas de dados**. Rio de Janeiro: Prentice Hall do Brasil, 1989.

Técnicas de programação recursiva

Você já deve ter visto, à venda em supermercados ou lojas de utilidades domésticas, um conjunto de travessas de diferentes tamanhos, mas com a mesma estrutura, encaixadas uma dentro da outra. Ou já deve ter notado que, quando se posiciona um espelho em frente a outro, a imagem refletida se torna infinita e menor a cada reflexo, não passando de um pixel ao final. Esse processo que ocorre em ambos os exemplos é chamado de recursão. A recursão está presente em diversas situações do dia a dia, principalmente em princípios matemáticos e, por consequência, no mundo dos algoritmos computacionais.

Algoritmos recursivos são programas com chamadas hierárquicas nas quais um método chama a ele próprio em sua implementação. A

recursão pode ocorrer de maneira direta ou indireta, sendo a segunda executada por um método intermediário (DEITEL; DEITEL, 2016).

Ao longo deste capítulo, serão apresentados os conceitos principais de um algoritmo e como ele pode ser utilizado para deixar o código mais elegante e de fácil compreensão. Também abordaremos o que deve ser evitado na definição de um algoritmo recursivo, sem que ele perca alguns de seus benefícios.

1 Definição da recursão

O primeiro ponto relevante ao se trabalhar com recursão é saber que ela somente pode resolver um caso simplificado do problema em si, que é chamado de caso básico. Sempre que o método recursivo encontrar um caso básico do problema, ele consegue retornar um resultado definitivo. Como consequência, os demais casos, os casos recursivos, obtêm sua resolução. Assim, toda vez que um problema complexo é apresentado, este deve ser dividido em pequenos subproblemas, mais simples, para que o problema original seja resolvido. Os subproblemas devem manter a mesma estrutura do original, sendo apenas um pouco mais simples ou menores, em termos de processamento (DEITEL; DEITEL, 2016).



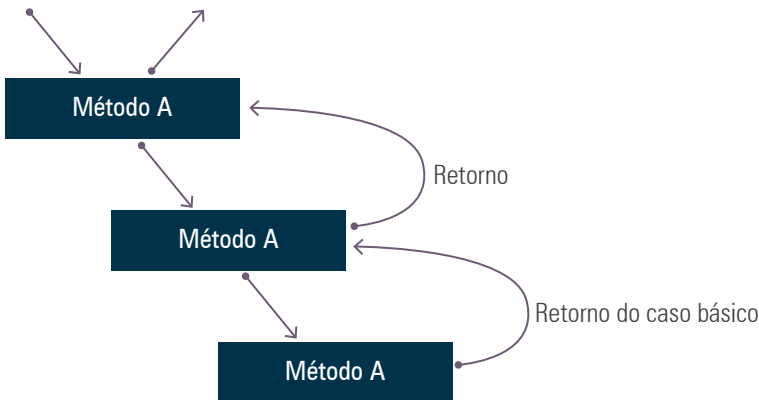
IMPORTANTE

Ao analisar problemas complexos e algoritmos com muitos laços aninhados, estes podem ter uma solução recursiva mais simples do que a solução inicial (GOODRICH; TAMASSIA, 2013).

O passo recursivo geralmente possui uma instrução do tipo *return*, cuja execução se encerra retornando o valor do resultado do processamento, o que torna possível a combinação entre os subproblemas ao obter o resultado do caso básico (WIRTH, 1989; DEITEL; DEITEL, 2016).

Esse passo é executado sempre que um método realiza uma chamada a ele mesmo na sua implementação. Assim como um algoritmo iterativo utiliza laços de repetições, a recursão também deve possuir um critério de parada, que determinará o fim de sua execução. Para que isso aconteça, cada chamada recursiva deve ser realizada com uma versão mais simples que o problema original. A versão mais simples, feita em cada passo da execução, levará o algoritmo a atingir o caso básico. Sendo assim, uma solução é encontrada e retornada para o resultado final (DEITEL; DEITEL, 2016). Veja na figura 1 a ilustração de uma recursão simples.

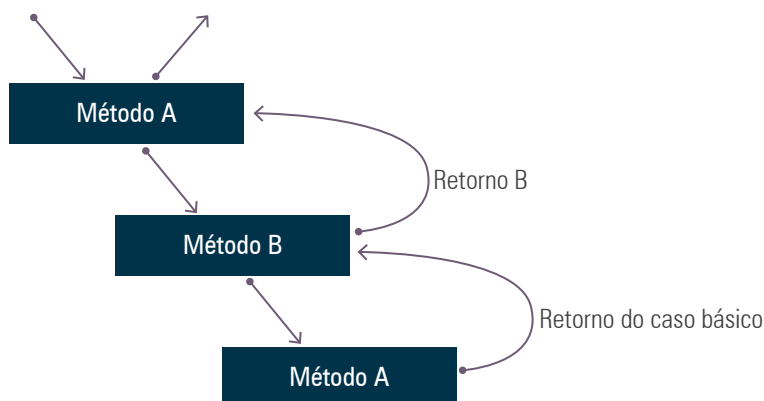
Figura 1 – Exemplo de recursão direta ou simples



Além de o método realizar uma chamada a si próprio, a recursão direta, pode ocorrer em um algoritmo o que se chama de recursão indireta. Esse tipo de recursão é feita através de um método intermediário. O funcionamento ocorre da seguinte maneira: o método A (recursivo) realiza a chamada a um método B (intermediário). Na implementação do método B, existe uma nova chamada recursiva ao método A. Ao observar esse fluxo, nota-se que, mesmo com a nova chamada ao método A através de B, a primeira chamada de A continua em aberto e aguardando o resultado retornado de B. Nesse momento, ocorre o processo recursivo, porém com um segundo método intermediando. Por isso, o

nome “recursão indireta” (GOODRICH; TAMASSIA, 2013; DEITEL; DEITEL, 2016). Veja a ilustração de uma recursão indireta na figura 2.

Figura 2 – Exemplo de recursão indireta



Em problemas complexos, a aplicação de recursão na solução traz o benefício de deixar o algoritmo legível e ainda a possibilidade de se manter uma solução eficiente em termos de processamento (GOODRICH; TAMASSIA, 2013).

Um exemplo de aplicação da recursão em computação é o sistema de arquivos existente em qualquer sistema operacional. Este é iniciado por um diretório ou pasta principal. Nele são armazenados arquivos e novos diretórios (recursão), os quais também possuem outros arquivos e diretórios. Esse processo ocorre até que o caso básico seja obtido. Na questão do sistema de arquivos, o caso básico é um diretório que possui em sua raiz apenas arquivos armazenados (GOODRICH; TAMASSIA, 2013; DEITEL; DEITEL, 2016).

2 Traga a matemática à vida

Neste tópico, apresentaremos alguns conceitos matemáticos que possuem definições recursivas, bem como suas respectivas implementações,

utilizando a linguagem de programação Java. Em alguns momentos, dicas para melhorar o desempenho do algoritmo nessa linguagem serão demonstradas.

2.1 Fatorial: o clássico da recursão

O problema do cálculo de um número fatorial é bem discutido em qualquer curso da área de exatas. A função fatorial é definida formalmente assim: dado um número inteiro e positivo chamado n , seu fatorial é obtido pelo produto de todos os números inteiros entre n e 1 (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Em termos matemáticos, a definição é:

$$n! = \prod_{k=1}^n k \quad \forall n \in \mathbb{N}$$

Em que a função fatorial é definida por um número seguido de um ponto de exclamação. Seu resultado é dado pelo produto da variável k iniciando em 1 e percorrendo até n . Isso acontece para todo n pertencente ao conjunto dos números naturais. Veja o exemplo de $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Perceba que o $5!$ pode ser definido de maneira recursiva, na qual $5! = 5 \cdot 4!$. A regra da função fatorial é então definida da seguinte maneira:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

O caso básico da função recursiva fatorial ocorre quando o valor de n é igual a 0. Nesse caso, a função fatorial retorna 1 por ser um valor indiferente à operação de multiplicação. Para todos os outros casos, em passos recursivos, o retorno é n multiplicado pelo valor do fatorial de $n - 1$. Veja a implementação da classe `FuncaoFatorial`:

```

1 public class FuncaoFatorial {
2     // função recursiva
3     public static int fatorial(int n) {
4         if(n <= 1) return 1; // caso básico
5         return n * fatorial(n - 1); // passo
recursivo
6     }
7
8     public static void main(String[] args) {
9         // cálculo do fatorial de 0 a 5
10        for(int n = 0; n <= 5; n++)
11            System.out.printf("%d! =>
%d\n", n, fatorial(n));
12    }
13 }

```

A classe implementada apresenta dois métodos. O primeiro, definido entre as linhas 3 e 6, é o método recursivo para o cálculo do fatorial de um número n qualquer. Na linha 4, é determinado o caso básico que verifica se n é menor ou igual a 1, retornando assim o próprio valor 1 (o menor valor de um fatorial). Na linha 5, é retornado o valor de n multiplicado pelo resultado da chamada recursiva para o fatorial de $n - 1$. O método *main*, definido entre as linhas 8 e 12, apenas executa a chamada para o cálculo dos fatoriais de 0 até 5. O resultado é apresentado abaixo:

```

0! => 1
1! => 1
2! => 2
3! => 6
4! => 24
5! => 120

```

Perceba que foi utilizado nas declarações de métodos e variáveis o tipo *int*. Isso pode gerar um problema de memória no cálculo fatorial. A partir do 12!, a variável do tipo *int* não consegue mais armazenar valores, ocorrendo um estouro de memória. A solução paliativa é efetuar a troca

para variáveis do tipo *long*, mas perceba que, desse modo, só será possível efetuar o cálculo até o 21!. Após isso, o estouro de memória ocorre novamente. No Java, existem duas classes que podem auxiliar em cálculos de problemas que possuem a precisão arbitrária e não podem ser feitos através dos tipos primitivos. São elas: *BigInteger* e *BigDecimal*, ambas encontradas no pacote *java.math* (DEITEL; DEITEL, 2016).



PARA SABER MAIS

Para compreender melhor os objetos *BigInteger* e *BigDecimal* utilizados nas implementações, acesse os links a seguir:

- <https://docs.oracle.com/javase/8/docs/api/java/math/class-use/BigInteger.html> (acesso em: 20 abr. 2020).
- <https://docs.oracle.com/javase/8/docs/api/java/math/class-use/BigDecimal.html> (acesso em: 20 abr. 2020).

Agora, acompanhe a reimplementação do algoritmo recursivo fatorial utilizando *BigInteger* como tipo:

```
1  import java.math.BigInteger;
2
3  public class FuncaoFatorial {
4      // função recursiva
5      public static BigInteger fatorial(BigInteger n) {
6          if(n.compareTo(BigInteger.ONE) <= 0) // caso
básico
7              return BigInteger.ONE;
8          return n.multiply( // passo recursivo
9              fatorial(n.
subtract(BigInteger.ONE)));
10     }
11 }
```

```

12     public static void main(String[] args) {
13         // cálculo do fatorial de 0 a 5
14         for(int n = 0; n <= 5; n++)
15             System.out.printf("%d! => %d\n", n,
16                               fatorial(BigInteger.
valueOf(n)));
17     }
18 }

```

Algumas particularidades sobre o código refatorado: (a) por não se tratar de um tipo primitivo, o uso dos operadores comuns não é possível nesse caso. (b) Na linha 6, o método *compareTo* retorna -1 se n for menor que 1, e nesse caso foi utilizada a constante que representa o valor 1 para o objeto *BigInteger*. Também retorna 0 se forem iguais e 1 se for maior. (c) Nas linhas 8 e 9, utilizam-se os métodos *multiply* e *subtract* para calcular o fatorial de n .



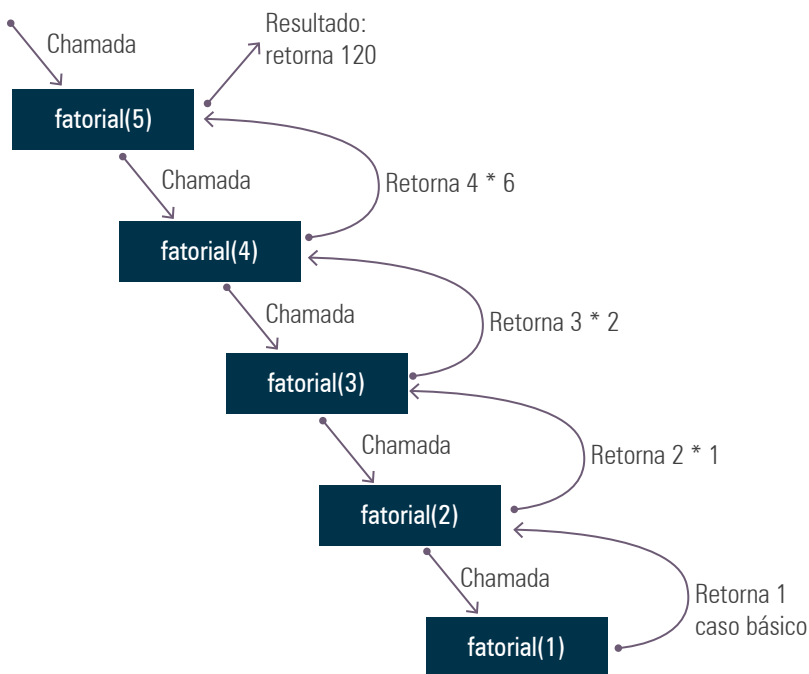
IMPORTANTE

Um erro muito comum ao trabalhar com algoritmos recursivos é omitir o caso básico. Esse erro pode levar a um problema chamado de recursão infinita. É um erro parecido com o do *loop* infinito em laços de repetição. Contudo, a recursão infinita pode gerar um estouro de memória de maneira muito mais rápida que um laço de repetição (DEITEL; DEITEL, 2016).

Entender um algoritmo recursivo muitas vezes não é uma tarefa trivial e conseguir visualizar todos os caminhos tomados pelo algoritmo pode ser algo bem complicado. Uma ferramenta que pode auxiliar nessa tarefa é chamada de rastreamento recursivo (GOODRICH; TAMASSIA, 2013). Basicamente, essa ferramenta desenha um quadro para cada chamada recursiva, conectando-as através de uma seta reta que representa cada chamada. No final, uma seta curva é retornada ao quadro

acima para representar o resultado do valor obtido naquele passo. A figura 3 apresenta o rastreamento recursivo para o fatorial do número 5:

Figura 3 – Rastreamento recursivo da função fatorial para o número 5



Na figura 3, ocorre uma chamada (seta direcional) de um método qualquer para a função fatorial, passando o valor 5 como parâmetro (retângulo representando o método). A função fatorial realiza uma nova chamada para ela mesma passando como parâmetro agora o valor 4, representando a subtração do número informado inicialmente em uma unidade. Esse processo se repete até que a função fatorial execute uma chamada com o parâmetro de valor 1. Nesse momento, pela implementação realizada, atinge-se o caso básico. Isso significa que as chamadas recursivas foram finalizadas. Então, retornam os resultados de cada método para o método anterior que realizou a chamada. Inicia-se o retorno do próprio valor 1; o retorno é o resultado da operação de multiplicação de 1 por 2, representando o fatorial de 2. Isso se repete até chegar

ao primeiro nível de chamada, o fatorial de 5. O resultado esperado é 120, passado ao programa principal ou ao método que realizou a primeira chamada para a função fatorial. Apesar de ser um modelo de entrada em algoritmos recursivos, o fatorial não é a única aplicação para essa técnica. Algumas outras aplicações serão abordadas na sequência.

2.2 A beleza de Fibonacci

A série de Fibonacci é uma sequência de números muito utilizada em diversas áreas, envolvendo principalmente arte e computação. Acredita-se que ela seja a medida para a beleza perfeita. Já na natureza, a série de Fibonacci é utilizada para descrever o comportamento de uma espiral. Ela se inicia com 0 e 1, sendo cada número subsequente o cálculo da soma entre os dois anteriores. A sequência converge para um valor constante de 1,618, aproximadamente. Essa constante é chamada de média áurea ou relação áurea (DEITEL; DEITEL, 2016).

Sua definição formal é:

$$fibonacci(n) \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & \text{para todo } n > 1 \end{cases}$$

Perceba que existem dois casos básicos no cálculo do número de Fibonacci. Os casos básicos são exatamente o resultado de *fibonacci(1)* e *fibonacci(0)*, que retornam o mesmo valor de *n* recebido, 1 e 0, respectivamente. Acompanhe a implementação a seguir que utiliza a chamada do método de Fibonacci para o cálculo dos representantes de número 0 até 10.

```
1. import java.math.BigInteger;
2.
3. public class NumeroFibonacci {
4.
```

```

5.      // Constante de apoio
6.      private static final BigInteger TWO = BigInteger.
valueOf(2);
7.
8.      // Declaracao do método de fibonacci recursivo
10.     public static BigInteger fibonacci(BigInteger n) {
11.         // Casos básicos
12.         if (n.equals(BigInteger.ZERO) ||
13.             n.equals(BigInteger.ONE)) {
14.             return n;
15.         }
16.         // Chamada recursiva de Fibonacci
17.         return fibonacci(n.subtract(BigInteger.ONE))
18.             .add(fibonacci(n.
subtract(TWO)));
19.     }
20.
21.     public static void main(String[] args) {
22.         for(int i = 0; i <= 10; i++)
23.             System.out.printf("Fibonacci(%d) =>
%d\n", i,
24.                               fibonacci(BigInteger.
valueOf(i)));
25.     }
26. }

```

A implementação do cálculo do valor de Fibonacci utiliza o objeto do tipo `BigInteger` para evitar o problema de armazenamento na memória do computador, como o cálculo do fatorial anteriormente apresentado. Analisando alguns pontos importantes na implementação, nota-se que nas linhas 17 e 18 o método de Fibonacci é invocado recursivamente duas vezes. Como curiosidade, a chamada da linha 24 não é recursiva e é a chamada que origina o processo de recursão. O resultado apresentado pela implementação é:

```

Fibonacci(0) => 0
Fibonacci(1) => 1
Fibonacci(2) => 1
Fibonacci(3) => 2
Fibonacci(4) => 3

```

```
Fibonacci(5) => 5  
Fibonacci(6) => 8  
Fibonacci(7) => 13  
Fibonacci(8) => 21  
Fibonacci(9) => 34  
Fibonacci(10) => 55
```

Ao invocar duas vezes o método, é necessário que haja um cuidado redobrado sobre o consumo de memória gerado por um método recursivo. Para exemplificar melhor essa situação, observe que, ao solicitar o cálculo do valor de Fibonacci para 30, são efetuadas 2.692.537 chamadas. O valor para 32 é calculado com 7.049.155 chamadas (DEITEL; DEITEL, 2016). Perceba na diferença entre os dois números que a quantidade de chamadas praticamente triplicou ao longo do processo. Isso significa que a quantidade de chamadas é $nk > 2k/2$, em outras palavras, o número de chamadas é exponencial em relação a k (GOODRICH; TAMASSIA, 2013).

Com todas essas informações é possível dizer que a implementação de Fibonacci através de recursão não deve ser considerada. Um método recursivo ocupa mais memória que um método iterativo, uma vez que para cada chamada do método são criadas novas instâncias de todas as informações presentes nele. Isso pode gerar um estouro de memória mais rápido do que um simples laço de repetição infinito. Nos tópicos seguintes, serão discutidos os tipos de recursão mais comuns, neles apresentamos uma forma de solucionar o problema de memória discutido no algoritmo recursivo de Fibonacci.

3 Tipos de recursão

Existem basicamente três tipos de recursão. São eles: linear, binária e múltipla. Ao longo do capítulo, alguns exemplos de cada um dos tipos serão apresentados para auxiliar na compreensão e principalmente para explicar como utilizar cada um deles na solução de problemas.

3.1 Recursão linear

Pode-se afirmar que um problema é resolvido por uma recursão linear quando um método possui apenas uma chamada recursiva em sua execução (GOODRICH; TAMASSIA, 2013). Um exemplo já apresentado aqui é a função fatorial. Um outro problema resolvido a partir de uma recursão linear é o somatório de um vetor. Esse é um problema bem simples que poderia ser resolvido de maneira iterativa, mas, como processo didático para a compreensão da recursão linear, ele será apresentado como um algoritmo recursivo.

A entrada do problema será um vetor v e um inteiro n , sendo $n \geq 1$. A implementação é apresentada a seguir:

```
1.  public class Somatorio {
2.      // Recursão linear para função do somatório
3.      public static int somatorio(int[] v, int n) {
4.          // Caso básico
5.          if (n == 1) return v[0];
6.          // Passo recursivo
7.          return somatorio(v, --n) + v[n];
8.      }
9.      // Aplicacao da função
10.     public static void main(String[] args) {
11.         int[] v = {1, 3, 5, 7, 9, 11, 13};
12.         System.out.printf("O valor do somatório é
13.         %d\n",
14.                             somatorio(v, v.length));
15.     }
16. }
```

No algoritmo, são trabalhados valores do tipo inteiro, pois não existe a necessidade de se armazenar números muito grandes nesse problema. Em um caso mais complexo, valores do tipo *long* já seriam suficientes para se trabalhar com ele. Da linha 3 até a linha 9, é implementada a função de somatório recursiva. Ela recebe como parâmetro os valores de v ,

que é um vetor de inteiros, e n , que representa o tamanho do vetor. A linha 5 apresenta a definição do caso base, que é exatamente quando resta apenas um elemento no vetor. Nesse momento, este elemento deve ser retornado. O passo recursivo encontra-se implementado na linha 8 do código apresentado. Nele, é implementada a chamada recursiva do método somatório, passando como parâmetro o vetor e n subtraído em 1 para representar o vetor diminuindo de tamanho e o próprio n .



PARA PENSAR

Ao desenvolver um código, é importante pensar em como o compilador irá lê-lo. Essa leitura sempre se inicia da esquerda para a direita. Em Java, utilizar o atalho de incremento (++) ou de decremento (--) tem efeitos distintos em relação ao posicionamento da variável associada. Caso a variável venha depois dessa instrução, a operação é realizada primeiro. Após a operação, o programa irá consultar o valor armazenado na variável, como no caso da linha 8 do algoritmo somatório utilizando recursão.

Temos uma consulta de valor primeiro à variável e em seguida o efeito da operação, caso o atalho de incremento ou decremento esteja posicionado depois da variável associada.

Em relação à recursão linear, ainda podemos apresentar o conceito de recursão final. A recursão final ocorre quando o último passo de um algoritmo recursivo é a chamada recursiva (GOODRICH; TAMASSIA, 2013). A somatória e o fatorial, apresentados anteriormente, não atendem a essa característica, pois ambos possuem uma operação matemática antes de realizar o retorno do método.

Uma das operações que podem ser exemplificadas como recursão final é a inversão de um vetor. Esse algoritmo deve receber um vetor, sua posição inicial e sua posição final. Após sua chamada, ele começa o processo de trocar os valores presentes nas posições inicial e final, e o processo se repete até que a posição inicial seja maior ou igual à posição final do vetor.

Confira a implementação do método *inverter* da classe *Vetor*:

```
1. import java.util.Arrays;
2.
3. public class Vetor {
4.     // Método auxiliar para troca de posição
5.     public static void troca(int[] v, int posA, int
posB) {
6.         int aux = v[posA];
7.         v[posA] = v[posB];
8.         v[posB] = aux;
9.     }
10.
11.     public static void inverter(int[] v, int inicio,
int fim) {
12.         if(inicio >= fim) return; //caso básico
13.         troca(v, inicio, fim);
14.         inverter(v, ++inicio, --fim); //passo
recursivo
15.     }
16.
17.     public static void main(String[] args) {
18.         int[] v = {1, 3, 5, 7, 9, 11, 13};
19.         inverter(v, 0, v.length - 1);
20.         System.out.println(Arrays.toString(v));
21.     }
22. }
```

O algoritmo recursivo está implementado entre as linhas 11 e 15. O ponto importante ocorre exatamente na linha 14, antes de o bloco ser concluído. Nesse ponto, nada além da chamada recursiva é executado. Essa é a principal característica para a definição de uma recursão final.

3.2 Recursão binária

A recursão é chamada de binária quando o algoritmo recursivo possui duas chamadas recursivas dentro de si (GOODRICH; TAMASSIA, 2013).

A essa altura, já sabemos que o cálculo do valor de Fibonacci, apresentado anteriormente, é um algoritmo de recursão binária. Notamos também que este não é um algoritmo eficiente, devido à quantidade de chamadas recursivas ao longo de seu processamento. Mas existem algoritmos de recursão binária que continuam eficientes durante todo o processo. Um exemplo simples é a somatória binária. Essa somatória altera um pouco a original, que apresentamos durante o tópico de recursão linear.

A primeira alteração são os parâmetros recebidos pelo método. Na versão linear, apenas o vetor e seu fim eram informados. Para a versão binária, é necessário informar, além do vetor, sua posição inicial e seu tamanho. O algoritmo irá dividir o vetor ao meio, até que este tenha o tamanho de um elemento apenas. Esse é o caso básico. No retorno dos passos recursivos, é feita a soma de cada um desses elementos para chegar ao resultado final do problema.

Veja a implementação da somatória binária a seguir:

```
1.  public class Somatorio {
2.      // Recursão binária para função do somatório
3.      public static int somatorio(int[] v, int i,int n) {
4.          // Caso básico
5.          if (n == 1) return v[i];
6.          // Passo recursivo
7.          return somatorio(v, i, (n + 1)/2)
8.              + somatorio(v, i + (n + 1)/2,
9.                  n/2);
10.     }
11.     // Aplicação da função
12.     public static void main(String[] args) {
13.         int[] v = {1, 3, 5, 7, 9, 11, 13};
14.         System.out.printf("O valor do somatório é
15.             %d\n",
16.                 somatorio(v, 0, v.length));
17.     }
```


A função recursiva é definida entre as linhas 3 e 9. Na linha 5, é verificado o caso básico, em que, se n (tamanho do vetor) for igual a 1, retorna-se o valor armazenado na posição i do vetor v . As linhas 7 e 8 estão com as chamadas recursivas da função. Como são duas as chamadas recursivas, trata-se de uma recursão binária.

Na primeira chamada, são passados o valor de i (início do vetor) e a posição referente ao meio do vetor. O valor obtido deve ser arredondado para cima, atingindo assim o teto do valor decimal. A operação matemática pra realizar esse arredondamento é $(\text{número} + (\text{divisor} - 1))/\text{divisor}$; como o divisor é 2, um atalho foi adotado ao somar 1 diretamente em n para obter o resultado desejado. Na chamada recursiva apresentada na linha 8, o meio arredondado para cima é informado como o início do vetor, e o meio com arredondamento para baixo define o tamanho do vetor. Assim, os vetores são divididos até que reste apenas um elemento e este seja retornado somando com o elemento da chamada ao lado.

3.3 Recursão múltipla

A recursão múltipla ocorre quando um método necessita realizar várias chamadas dentro dele próprio (GOODRICH; TAMASSIA, 2013). Quando um método de recursão binária é apresentado, seu espaço de memória pode facilmente chegar a um problema exponencial, assim como ocorre em Fibonacci. Uma recursão múltipla é bem mais do que o dobro de chamadas recursivas, então é sempre bom analisar o problema para tentar diminuir sua complexidade ao final.

Esse tipo de recursão é muito utilizado em soluções de jogos conhecidos como *puzzles*, que possibilitam a análise de diversos caminhos que podem ser seguidos para se encontrar a solução.

4 Iteração ou recursão? Eis a questão

Quando se analisa um problema, tanto de maneira iterativa como por meio da recursão, deve-se ter em mente que ambos possuem sua base em uma instrução de controle. A instrução de controle pode ser chamada de critério de parada, no caso dos laços de repetição como *for*, *while* e *do-while*. Já para os casos recursivos, as instruções de controle são conhecidas como caso básico.

A instrução de controle serve para auxiliar o término do procedimento, assim é possível evitar que entrem no chamado *loop* infinito ou em recursão infinita. O objetivo de cada uma é um pouco diferente quando pensamos em sua construção. A versão iterativa utiliza uma variável de controle para verificar quando o algoritmo chegou ao fim, por exemplo, um contador de passos executados. A versão recursiva sempre divide o problema em versões mais simples e menores para que possa ser resolvido de maneira mais fácil.

Quando a solução é iterativa, ela geralmente consome menos memória que uma versão recursiva. Contudo, muitas vezes ela pode ser uma solução muito complexa de ser implementada e, às vezes, até mesmo de ser encontrada. A solução recursiva pode compensar o problema de memória utilizando uma infraestrutura mais avançada como um *cluster* de computadores. Assim, ela pode processar cada chamada recursiva em computadores diferentes, melhorando o desempenho do processamento. Encontrar uma solução recursiva é mais fácil e muitas vezes mais elegante que a iterativa.

Então qual deve ser escolhida? Não existe uma resposta exata a esta pergunta, mas sim uma recomendação de boa prática. Se a solução iterativa for tão clara quanto a recursiva, opte sempre por ela. A recursão foi desenvolvida para simplificar problemas maiores que não são trivialmente solucionados por um algoritmo iterativo.

5 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Implemente o fatorial de maneira iterativa e compare com a implementação recursiva. Utilize o gerenciador de processos para comparar o consumo de memória e o processamento.
2. Implemente o cálculo de Fibonacci de maneira iterativa e compare com a implementação recursiva. Utilize o gerenciador de processos para comparar o consumo de memória e o processamento.
3. A implementação recursiva de Fibonacci mais facilmente encontrada é a binária. Porém, o problema de Fibonacci é um problema linear. Dessa maneira, pode-se deduzir que sua implementação pode ser realizada de maneira recursiva linear. Faça a implementação.
4. Implemente a busca sequencial de maneira recursiva, se possível.
5. Implemente a busca binária de maneira recursiva, se possível.
6. Resolva de maneira recursiva um produtório, a partir de valores armazenados em um vetor.
7. Resolva o produtório como uma recursão binária.

Considerações finais

Neste capítulo, foram apresentados os conceitos de recursão, suas vantagens e desvantagens. Problemas matemáticos foram apresentados para demonstrar como questões do dia a dia já utilizam a teoria de recursividade para serem resolvidas. Questões importantes relacionadas ao consumo de memória, entre outros fatores, foram

apresentadas para auxiliar a tomada de decisão sobre como e quando utilizar um algoritmo recursivo.

Alguns exemplos foram implementados para apresentar os tipos de recursão e quais são os pontos de atenção para cada um deles. Por fim, uma comparação com algoritmos iterativos foi apresentada para auxiliar a tomada de decisão para o uso de algoritmos em projetos complexos. É importante compreender o problema a ser resolvido para se adotar a melhor estratégia para realizar sua implementação.

Referências

DEITEL, Paul J.; DEITEL, Harvey M. **Java: como programar**. 10. ed. São Paulo: Pearson Education, 2016.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: MAKRON Books, 2004.

WIRTH, Niklaus. **Algoritmos e estruturas de dados**. Rio de Janeiro: Prentice Hall do Brasil, 1989.

Algoritmos de ordenação simples

Entrar em um local desorganizado com o desejo de encontrar algo específico pode ser uma missão quase impossível. Muitas vezes, horas são perdidas para encontrar o que se procura. Por outro lado, quando existe uma determinada organização no ambiente, essa mesma tarefa acaba sendo prática, rápida e eficiente.

Em computação, o cenário descrito também ocorre. Quando os dados se mantêm ordenados, encontrar um determinado objeto é mais rápido, pois, para isso, podem ser utilizados algoritmos como o da busca binária, que encontra um elemento em, aproximadamente, metade do tempo dos demais algoritmos.

O problema de ordenação, em computação, não é só estudado por questões de organização e ordenação de dados. As possibilidades de pesquisas sobre esses algoritmos vão muito além disso. As funções apresentadas na construção dos algoritmos de ordenação e as técnicas utilizadas tornam-se referência para a solução de diversos outros problemas, principalmente em áreas como inteligência artificial. Outro ponto interessante é o seu uso didático na solução de problemas. Apesar de ser uma ciência exata, solucionar problemas computacionais pode apresentar muitas alternativas. Algoritmos de ordenação oferecem múltiplas soluções para um mesmo problema (WIRTH, 1989).

Graças às diversas possibilidades de solução, o uso dos algoritmos de ordenação vai além do simples fato de ordenar. Esse tipo de algoritmo tornou-se um *benchmark* (teste realizado para classificar os melhores métodos de acordo com um parâmetro) para comparações e análises de complexidade e de desempenho dos algoritmos (WIRTH, 1989). Existem duas classificações de aplicação para os algoritmos de ordenação: memória interna e memória externa.

Algoritmos que trabalham com memória interna geralmente ordenam estruturas de dados alocadas nas memórias do computador, como a RAM e a cache. Tendem a ser algoritmos mais rápidos, já que o acesso a esse tipo de memória é aleatório e a velocidade de leitura e de escrita é bem maior do que a de um disco rígido, por exemplo. Já os algoritmos de memória externa são mais lentos, pois são utilizados na organização de arquivos que, por sua vez, estão armazenados em dispositivos físicos como discos rígidos, SSDs e mídias. Outro aspecto ao qual se atribui a lentidão dos algoritmos de memória externa é que o

acesso a esse tipo de dispositivo é feito de maneira sequencial, impossibilitando algumas técnicas de programação em memórias de acesso aleatório. Além disso, dispositivos físicos possuem uma limitação de tamanho muito maior que as memórias RAM, por exemplo, o que torna o volume de dados muito grande, mesmo para uma ordenação simples (WIRTH, 1989).

Neste capítulo, trataremos apenas dos algoritmos de memória interna. Entre os mais comuns, existem diversas técnicas que são utilizadas para atingir o objetivo de ordenação de dados ou informações. Os algoritmos que apresentam as técnicas mais simples são o bubble sort, ou ordenação por bolhas, o insertion sort e o selection sort. Eles serão apresentados e discutidos nos tópicos a seguir.

1 Bubble sort

Dentre os métodos de ordenação existentes, o mais simples de ser implementado é o de ordenação por bolhas. É possível encontrar também, na literatura, esse método denominado como classificação por bolhas ou bubble sort, nome em inglês. O método leva esse nome pois o processo todo do algoritmo se assemelha a bolhas de gás de um refrigerante (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Quando uma bolha está leve, ela vai trocando de posição com as bolhas mais pesadas até alcançar a borda do copo.

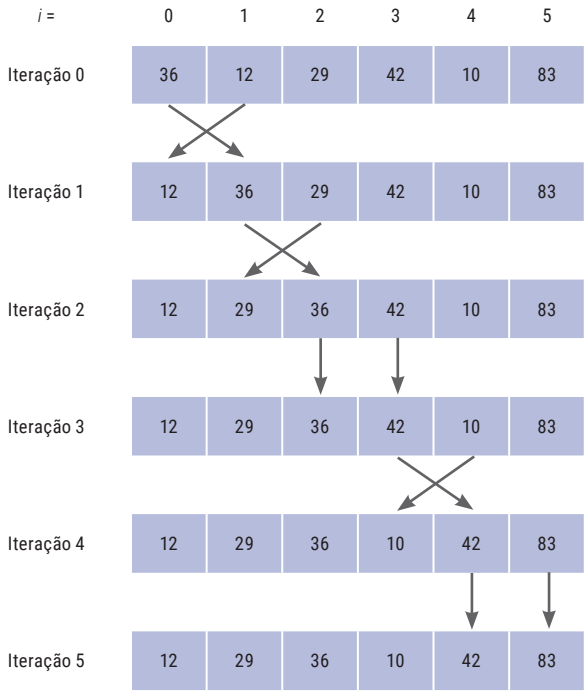
Por ter implementação e compreensão simples, geralmente, ele é o primeiro tipo a ser estudado pelos iniciantes em algoritmos. Contudo, a facilidade de implementação faz com que ele seja, provavelmente, o algoritmo de ordenação menos eficiente (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). O algoritmo funciona da seguinte maneira: ele percorre o vetor ou o arquivo sequencialmente por várias iterações. A cada iteração, ele compara o elemento em evidência com o seu sucessor. Em termos formais, temos $v[i]$ com $v[i + 1]$, em que v é o vetor ou o

arquivo em processo de ordenação. Após a comparação, se o sucessor for menor que o elemento $v[i]$, eles são trocados de posição. Quando a primeira passagem pelo vetor é concluída, o elemento $v[n - 1]$, sendo n a quantidade de elementos armazenados no vetor, encontra-se na posição correta (TENENBAUM; LANGSAM; AUGENSTEIN, 2004).

A figura 1, a seguir, apresenta a primeira passagem do algoritmo bubble sort por um vetor de inteiros com 6 elementos. O vetor apresentado possui 6 números inteiros na seguinte ordem [36, 12, 29, 42, 10, 83]. No seu início, é estabelecida a variável i , que controlará o índice atual da iteração do algoritmo. Os possíveis valores de i são apresentados acima da ilustração de cada posição do vetor correspondente. Logo abaixo está o primeiro passo da iteração, em que o valor de i é igual a 0. Sendo assim, existe uma comparação entre os valores de $v[0]$ e $v[1]$. Como $v[0]$ é maior que $v[1]$, é realizada uma troca entre esses dois elementos. Inicia-se então o valor de iteração 1. Agora, a comparação é entre $v[1]$ e $v[2]$, ocorrendo novamente a troca, pois o valor de $v[1]$ é maior. Imediatamente depois, é iniciada a iteração de número 2. Nessa iteração, os valores se mantêm em suas posições, porque o valor posterior $v[3]$ é maior que o atual $v[2]$. A próxima iteração é a de número 3, e nessa também ocorre uma troca. Na iteração 4, os valores são mantidos, finalizando o processo na iteração 5, na qual o maior valor entre os elementos é posicionado no final do vetor.

Nesse momento, pode-se considerar que o último elemento do vetor está posicionado no local correto de ordenação. Contudo, o algoritmo deve seguir as demais passagens até que todo o vetor esteja ordenado.

Figura 1 – Primeira passagem do bubble sort



Agora, entenda como é feita a implementação do bubble sort:

```
1 import java.util.Arrays;
2
3 public class BubbleSort {
4
5     public static void bubbleSort(int[] v) {
6         for (int i = 0; i < v.length; i++) {
7             for (int j = 0; j < v.length - 1;
8 j++)
9                 if (v[j] > v[j + 1])
10                     troca(v, j, j + 1);
11             System.out.printf("Passagem %d -> %s
12 \n",
13                               i, Arrays.toString(v));
14         }
15     }
16 }
```

```

13     }
14
15     public static void troca(int[] v, int a, int b) {
16         int aux = v[a];
17         v[a] = v[b];
18         v[b] = aux;
19     }
20
21     public static void main(String[] args) {
22         int[] v = {36, 12, 29, 42, 10, 83};
23         System.out.printf("Início -> %s \n",
24                             Arrays.toString(v));
25         bubbleSort(v);
26         System.out.printf("Fim -> %s \n",
27                             Arrays.toString(v));
28     }
29 }

```

Algumas considerações sobre o código-fonte apresentado:

- O método `troca` apresentado entre as linhas 15 e 19 foi criado para auxiliar no processo de troca dos valores de posição dentro do vetor apenas para facilitar e deixar o código mais legível.
- No método `bubble sort`, nas linhas de 5 a 13, foram necessários dois laços de repetição do tipo *for*. O primeiro, com a variável de controle *i*, representa a passagem pelo vetor. O *for* com a variável de controle *j* representa as iterações ilustradas na figura 1.
- Na linha 10, foi inserida uma saída do programa para verificar o resultado de cada passagem do algoritmo. Entretanto, no algoritmo original, não existe essa necessidade, ela foi utilizada apenas para fins didáticos.

Outro ponto importante sobre essa implementação é que, mesmo que o vetor esteja completamente ordenado, as passagens são executadas até o final. Realizar essas passagens com o vetor já ordenado apenas deixa o algoritmo mais ineficiente. Veja a saída da implementação e observe o resultado e o efeito mencionado:

```

Início      -> [36, 12, 29, 42, 10, 83]
Passagem 0 -> [12, 29, 36, 10, 42, 83]
Passagem 1 -> [12, 29, 10, 36, 42, 83]
Passagem 2 -> [12, 10, 29, 36, 42, 83]
Passagem 3 -> [10, 12, 29, 36, 42, 83]
Passagem 4 -> [10, 12, 29, 36, 42, 83]
Passagem 5 -> [10, 12, 29, 36, 42, 83]
Fim         -> [10, 12, 29, 36, 42, 83]

```

Na saída obtida, é possível observar que, a partir da passagem 3, o algoritmo já ordenou por completo o vetor. Nesse ponto, não há a necessidade de mais passagens. Outro ponto que pode ser melhorado é o número de iterações a cada passagem. Para isso, é preciso subtrair o valor da passagem do tamanho do vetor no segundo *for*. Esse procedimento fará com que os valores já ordenados, as últimas posições do vetor, não sejam mais comparadas. Todas as medidas tomadas a partir desse momento são para aumentar o desempenho do algoritmo.

A seguir, apresentamos a nova versão do algoritmo bubble sort:

```

1  public class BubbleSort {
2
3      public static void bubbleSort(int[] v) {
4          boolean troca = true;
5          for (int i = 0; i < v.length && troca; i++)
6          {
7              troca = false;
7              for (int j = 0; j < v.length - i - 1;
7              j++) {
8                  if (v[j] > v[j + 1]) {
9                      trocar(v, j, j + 1);
10                     troca = true;
11                 }
12             }
13             System.out.printf("Passagem %d -> %s\n",

```

```

14                                     i, Arrays.toString(v));
15     }
16 }
17
18 public static void trocar(int[] v, int a, int b) {
19     int aux = v[a];
20     v[a] = v[b];
21     v[b] = aux;
22 }
23
24 public static void main(String[] args) {
25     int[] v = {36, 12, 29, 42, 10, 83};
26     System.out.printf("Início      -> %s \n",
27                       Arrays.toString(v));
28     bubbleSort(v);
29     System.out.printf("Fim        -> %s \n",
30                       Arrays.toString(v));
31 }
32 }

```

Perceba que a implementação realizada apresentou uma variável a mais de controle (troca) e a iteração não é executada até o final do vetor, impedindo a comparação com as posições já ordenadas. Como o vetor do exemplo é pequeno e a quantidade de passagens também não é significativa, não é possível ver uma melhora expressiva. Contudo, quando o vetor estiver totalmente ordenado, ele fará apenas uma passagem pelo vetor e não mais n vezes n passagens como na versão anterior. Veja a saída da nova versão do método bubble sort:

```

Início      -> [36, 12, 29, 42, 10, 83]
Passagem 0 -> [12, 29, 36, 10, 42, 83]
Passagem 1 -> [12, 29, 10, 36, 42, 83]
Passagem 2 -> [12, 10, 29, 36, 42, 83]
Passagem 3 -> [10, 12, 29, 36, 42, 83]
Passagem 4 -> [10, 12, 29, 36, 42, 83]
Fim         -> [10, 12, 29, 36, 42, 83]

```

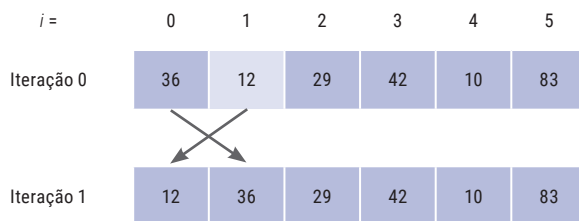
Perceba que, nesse caso, apenas uma passagem da versão inicial do algoritmo foi subtraída. Isso ocorre porque, na passagem 3, o algoritmo finaliza a ordenação efetivamente. Na passagem 4 não ocorre nenhuma troca, o que finaliza o processo de ordenação. Assim, a passagem 5, existente na versão inicial, não ocorre nessa nova versão. Em vetores com tamanhos maiores, é possível ter um ganho mais significativo.

2 Insertion sort

Imagine um jogo de pôquer: os jogadores recebem, cada um, um conjunto de cartas e as colocam nas mãos. A entrega das cartas é aleatória, portanto, sem nenhuma ordem específica. Para facilitar e agilizar suas jogadas, o jogador as organiza em ordem crescente da esquerda para a direita. Para ordenar as cartas, o jogador, geralmente, segura as cartas com uma das mãos e com a outra tira as cartas menores do final e as coloca na ordem correta a partir da carta menor até a carta maior. Esse processo é chamado de ordenação por inserção, ou insertion sort, em inglês (CORMEN *et al.*, 2002).

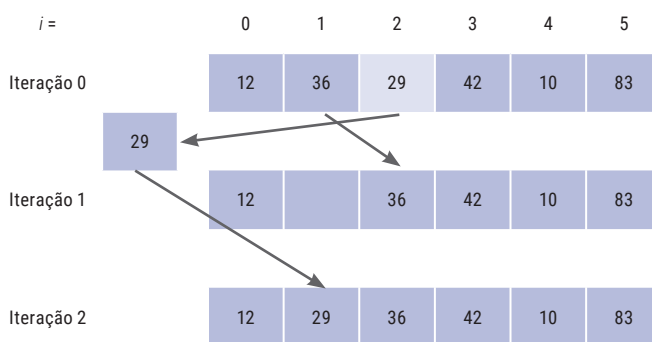
O insertion sort é um algoritmo de ordenação que efetua esse processo removendo os elementos menores do final do vetor e os reinserindo no início, já em sua posição final. O passo a passo do algoritmo começa com a seleção a partir da segunda posição do vetor. Ele copia esse elemento para uma variável auxiliar, depois ele o compara com todos os elementos à sua esquerda. Se ele for menor que o elemento à esquerda, ele continua procurando a posição onde deverá ser inserido. O critério para a posição é: elemento à esquerda menor; à direita, maior (CORMEN *et al.*, 2002; ASCENCIO; ARAÚJO, 2010). Veja na figura 2 o primeiro passo do algoritmo insertion sort:

Figura 2 – Primeira passagem do algoritmo insertion sort no vetor



Na figura 2, é apresentada a primeira passagem do algoritmo insertion sort. O elemento da segunda posição é selecionado (em cor mais clara). Nesse momento, ele é armazenado em uma variável auxiliar. A partir disso, ele será comparado com todos os elementos à esquerda e será posicionado no local de ordenação do vetor. Como nessa primeira iteração é realizada apenas uma comparação, a troca ocorre entre a primeira posição e a segunda, já que o elemento 12 (segunda posição) é menor que o elemento 36 (primeira posição). Para continuar o processo de compreensão do algoritmo, analise a figura 3. Ela apresenta a segunda passagem do algoritmo.

Figura 3 – Segunda passagem do algoritmo insertion sort no vetor



Na figura 3, é possível observar o passo a passo da segunda passagem do algoritmo insertion sort. Nesse momento, o elemento selecionado é o 29 (em cor mais clara). Ele é armazenado em uma variável separada, representada pelo quadrado flutuante entre as iterações 0 e 1, e, em seguida, é comparado com o primeiro elemento à sua esquerda, no caso o 36, que ocupa a posição de índice 1 no vetor. Como 36 é maior

que 29, ele é copiado para a posição de índice 2, ocupada antes pelo 29. Agora, o algoritmo compara a posição de índice 0 que tem o valor 12, com o 29. Como 12 é menor que 29, as comparações se encerram e o valor selecionado é copiado na posição vazia do vetor, que, nesse caso, é a 1. Veja a implementação do código em Java:

```
1  public class InsertionSort {
2
3      public static void insertionSort (int[] v) {
4          for (int i = 1; i < v.length; i++) {
5              int x = v[i];
6              for (int j = i - 1; j >= 0 && v[j] >
x; j--) {
7                  v[j + 1] = v[j];
8                  v[j] = x;
9              }
10             System.out.printf("Iteração do nro %d
-> %s \n",
11                             x, Arrays.toString(v));
12         }
13     }
14
15     public static void main(String[] args) {
16         int[] v = {36, 12, 29, 42, 10, 83};
17         System.out.printf("Início          -> %s
\n",
18                             Arrays.toString(v));
19         insertionSort(v);
20         System.out.printf("Fim            -> %s
\n",
21                             Arrays.toString(v));
22     }
23 }
```

O método de ordenação insertion sort é apresentado entre as linhas 3 e 13. Perceba que a variável *x* é a variável responsável por armazenar o elemento que será ordenado. Se fosse no jogo de pôquer, seria a carta que o jogador segurou em sua mão antes de decidir onde colocá-la. Na linha 6, existe um laço de repetição do tipo *for*, que é inicializado com o

primeiro elemento anterior a x , ou seja, à sua esquerda. Sendo decrescente esse passo, isso garante que apenas os elementos já ordenados serão percorridos. Para fins didáticos, na linha 10 foi inserido o passo a passo do algoritmo. Veja o resultado a seguir:

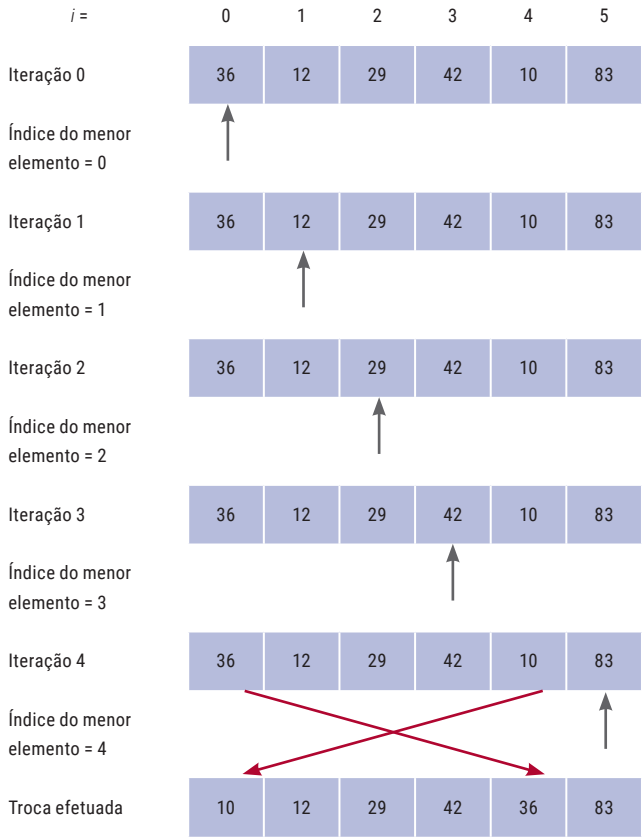
```
Início          -> [36, 12, 29, 42, 10, 83]
Iteração do nro 12 -> [12, 36, 29, 42, 10, 83]
Iteração do nro 29 -> [12, 29, 36, 42, 10, 83]
Iteração do nro 42 -> [12, 29, 36, 42, 10, 83]
Iteração do nro 10 -> [10, 12, 29, 36, 42, 83]
Iteração do nro 83 -> [10, 12, 29, 36, 42, 83]
Fim             -> [10, 12, 29, 36, 42, 83]
```

Perceba como os números em destaque na saída vão ficando ordenados a cada passagem do algoritmo, pois ele se preocupa apenas com os elementos à esquerda. No caso do número 42, por exemplo, ele já é maior que o 36 na primeira iteração. Isso faz com que ele seja mantido na posição e o algoritmo não tenha a necessidade de percorrer o vetor. Esse ponto é o que traz um diferencial em comparação com o bubble sort. O insertion sort, apesar de poder percorrer o vetor duas vezes, como o bubble sort, mantém-se com uma média de desempenho melhor (CORMEN *et al.*, 2002; ASCENCIO; ARAÚJO, 2010).

3 Selection sort

O algoritmo de ordenação por seleção, ou selection sort, em inglês, apresenta uma terceira estratégia para realizar o processo de ordenação de um vetor. Em linhas gerais, o selection sort percorre todo o vetor em busca do menor elemento não ordenado. Quando ele o encontra, ele posiciona o valor mínimo na primeira posição não ordenada do vetor (CORMEN *et al.*, 2002; ASCENCIO; ARAÚJO, 2010). Veja a primeira passagem do algoritmo para o vetor v , representado pela figura 4.

Figura 4 – Primeira passagem do algoritmo selection sort



O passo a passo da primeira passagem do algoritmo selection sort demonstra todo o funcionamento dele. Na primeira iteração, o índice do menor elemento é o próprio 0, pois é o primeiro elemento a ser comparado. Para a próxima iteração, o elemento que está na posição 1 é menor que o da posição 0, então o índice de menor elemento passa a ser 1. Na iteração 2, o valor 29 na posição 2 não é menor que o 12 na posição de menor elemento. O algoritmo mantém o índice escolhido. Na próxima iteração, de número 3, os valores também se mantêm, pois 42 é um número maior que 12. Seguindo para a iteração de número 4, existe uma atualização do índice de menor valor, pois 10 é menor que 12.

Sendo assim, o índice, nesse momento, passa a ser o de valor 4. Na última iteração, de número 5, não há nenhuma troca ou atualização, pois 83 é o maior valor do vetor do exemplo. Após esse passo, é realizada a troca do valor na posição 0 (36) com o da posição 4 (10). O algoritmo prossegue nesse passo a passo até que o vetor esteja completamente ordenado.

Agora que o passo a passo da execução do algoritmo foi estabelecido, deve-se implementar o método de ordenação. O exemplo a seguir é implementado utilizando a linguagem de programação Java:

```
1  public class SelectionSort {
2
3      public static void selectionSort(int[] v) {
4          for (int i = 0; i < v.length; i++) {
5              int sindex = i;
6              for (int j = i + 1; j < v.length;
7              j++) {
8                  if (v[j] < v[sindex]) sindex =
9                  j;
10                 trocar(v, i, sindex);
11                 System.out.printf("Iteração %d -> %s
12                 \n",
13                 i, Arrays.toString(v));
14             }
15         }
16     }
17
18     public static void trocar(int[] v, int a, int b) {
19         int aux = v[a];
20         v[a] = v[b];
21         v[b] = aux;
22     }
23
24     public static void main(String[] args) {
25         int[] v = {36, 12, 29, 42, 10, 83};
26         System.out.printf("Início -> %s \n",
27             Arrays.toString(v));
28         selectionSort(v);
29     }
30 }
```

```
26         System.out.printf("Fim      -> %s \n",  
27                             Arrays.toString(v));  
28     }  
29 }
```

Na implementação realizada, é necessário que haja novamente a existência do método *trocar* para auxiliar no momento da permuta dos valores para o menor índice, veja as linhas de 15 a 19. Entre as linhas 3 e 13, ocorre a implementação do algoritmo selection sort. No laço de repetição principal, é possível verificar que o vetor é percorrido de ponta a ponta. O índice do menor elemento é inicializado na linha 5, com o valor de *i* dado no laço de repetição. Na linha 6, inicia-se o segundo laço de repetição que percorre o vetor da posição *i* + 1 até o seu fim. Ao final, na linha 9, o método *trocar* é acionado. Na linha 10, para fins didáticos, é exibido o passo a passo do algoritmo após as trocas do vetor. O resultado é este:

```
Início      -> [36, 12, 29, 42, 10, 83]  
Iteração 0 -> [10, 12, 29, 42, 36, 83]  
Iteração 1 -> [10, 12, 29, 42, 36, 83]  
Iteração 2 -> [10, 12, 29, 42, 36, 83]  
Iteração 3 -> [10, 12, 29, 36, 42, 83]  
Iteração 4 -> [10, 12, 29, 36, 42, 83]  
Iteração 5 -> [10, 12, 29, 36, 42, 83]  
Fim        -> [10, 12, 29, 36, 42, 83]
```

No final, o selection sort apresentou o pior desempenho entre os três algoritmos apresentados, pois ele não possibilita nenhuma modificação para garantir um melhor desempenho de maneira trivial.

4 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Execute o algoritmo bubble sort para os vetores abaixo e compare as diferenças entre os resultados:
 - a. [36, 15, 75, 2, 16, 48, 42, 51]
 - b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
 - c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
2. Execute o algoritmo insertion sort para os vetores abaixo e compare as diferenças entre os resultados:
 - a. [36, 15, 75, 2, 16, 48, 42, 51]
 - b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
 - c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
3. Execute o algoritmo selection sort para os vetores abaixo e compare as diferenças entre os resultados:
 - a. [36, 15, 75, 2, 16, 48, 42, 51]
 - b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
 - c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
4. Compare os resultados dos três algoritmos.

Considerações finais

Neste capítulo, foi possível perceber como existem diversas soluções para um mesmo problema. Os algoritmos demonstrados apresentam, em média, o mesmo desempenho computacional. Alguns deles podem ser otimizados por meio de truques em sua implementação e

até mesmo pelas próprias linguagens de programação. Esses algoritmos são os mais comuns e os mais fáceis de se implementar. Existem outros mais avançados, com técnicas que permitem que sejam muito mais rápidos que os apresentados aqui, mas eles serão abordados no próximo capítulo.

Referências

ASCENCIO, Ana Fernanda Gomes; ARAÚJO, Graziela Santos de. **Estruturas de dados**: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010.

CORMEN, Thomas H. *et al.* **Algoritmos**: teoria e prática. Rio de Janeiro: Editora Campus, 2002.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: MAKRON Books, 2004.

WIRTH, Niklaus. **Algoritmos e estruturas de dados**. Rio de Janeiro: Prentice Hall do Brasil, 1989.

Algoritmos de ordenação sofisticados

Com a evolução dos estudos relacionados a algoritmos, muitas técnicas vão se complementando e algumas novas podem surgir. O objetivo é melhorar continuamente o desempenho nas soluções de problemas, tanto nos conhecidos como nos desconhecidos. Em problemas de ordenação, pode-se encontrar uma grande fartura de novas técnicas e ainda obter um bom ambiente para compará-las. Algoritmos de ordenação são explorados nessas situações quando se tem um conhecimento claro do problema a ser resolvido. Isso torna a classificação das técnicas empregadas nos algoritmos o trabalho mais realizado entre os pesquisadores.

Uma das técnicas mais utilizadas para solucionar problemas complexos e que exigem um grande número de comparações é a técnica de dividir e conquistar, que consiste em pegar um problema relativamente grande e quebrá-lo em subproblemas menores até que sua resolução seja praticamente uma única operação computacional (SEdgeWICK;

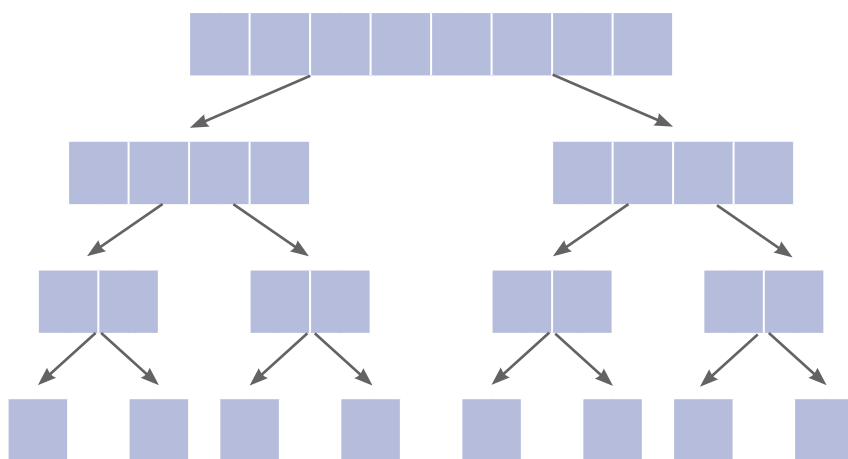
FLAJOLET, 2013). Os subproblemas obtidos através da divisão do problema original devem estar intimamente relacionados a ele (CORMEN *et al.*, 2002). Diversos algoritmos de ordenação utilizam essa técnica, e por isso são considerados algoritmos de ordenação sofisticados.

A técnica de dividir e conquistar é definida com três passos (CORMEN *et al.*, 2002; GOODRICH; TAMASSIA, 2013):

- **Dividir:** o problema é dividido em um determinado número de subproblemas.
- **Conquistar:** os subproblemas são resolvidos recursivamente.
- **Combinar:** junta-se as soluções dos subproblemas combinando-os e obtendo ao final a solução do problema original.

Dividir é o passo mais simples para a maioria dos problemas que serão resolvidos com essa técnica. A figura 1 ilustra o processo de divisão de um vetor com oito posições.

Figura 1 – Processo de divisão de um problema com um vetor de 8 posições, até que ele seja quebrado em 8 subproblemas



Note que o processo de divisão é naturalmente obtido através de um algoritmo recursivo a partir do problema original, que é dividido até chegar a seu caso básico. Após o processo de divisão, o caso básico é concluído, obtendo assim o resultado de cada um dos subproblemas definidos no processo da divisão. Com os resultados dos casos básicos calculados, o processo de conquista também é finalizado. Nesse momento, inicia-se o processo de combinação, que, usando como exemplo o somatório das raízes quadradas, será a soma dos resultados individuais de cada elemento, no momento de retorno de cada chamada recursiva do problema.

A técnica de divisão e conquista é por si mesma recursiva, como observado no exemplo da figura 1. Sendo assim, descobrir de forma precisa seu desempenho é uma tarefa não trivial. Um dos motivos é porque nem sempre a divisão dos problemas será exata, já que em muitos casos os conjuntos classificados podem conter uma quantidade ímpar de elementos. Isso gera uma divisão não muito precisa para se contabilizar de maneira simétrica ambas as partes divididas (SEdgeWICK; FLAJOLET, 2013).

Distinguir a diferença de desempenho quando o total de elementos é ímpar ou par não é uma tarefa precisa quando o conjunto de dados contém milhares de elementos. Porém, em conjuntos menores, a diferença no desempenho é notável. Essa diferença pode ser expressiva se a divisão do subproblema for maior que 2 a cada chamada recursiva. Isso ocorre porque existe uma variação na quantidade de subproblemas chamados de maneira recursiva. Ao juntá-los para a solução do problema original, o custo pode ser alto e ainda pode ocorrer sobreposição de subproblemas, gerando resultados ambíguos (SEdgeWICK; FLAJOLET, 2013).

Contudo, o uso da técnica de divisão e conquista é muito ampla em diversas aplicações. Ela é uma técnica elegante e permite simplificar muito o código da solução, já que utiliza a recursão para isso. Ao longo deste capítulo, serão explorados dois algoritmos de ordenação. Estes são definidos de maneira a aplicar a técnica de dividir e conquistar. São eles: o merge sort e o quicksort.

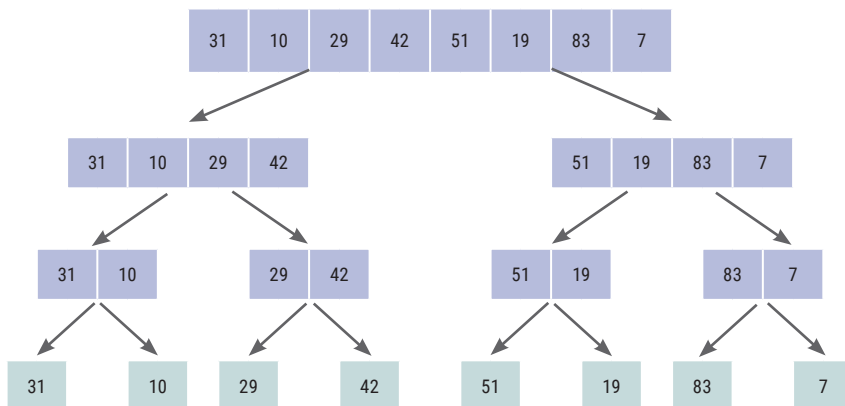
1 Merge sort

Dentre os algoritmos de ordenação que aplicam a técnica de divisão e conquista, o que possui o processo mais simples e eficiente sem dúvidas é o merge sort. Ele é um algoritmo recursivo que se divide em subvetores até obter sua ordenação. Como o merge sort é um processo recursivo, é necessário estabelecer as regras para os casos básico e recursivo (GOODRICH; TAMASSIA, 2013). Veja as regras:

- **Caso básico:** se o vetor ou subvetor possuir apenas um único elemento, ou, se ele for vazio, retornará o próprio vetor, e assim ele será considerado ordenado, uma vez que não existe uma falta de ordem quando há apenas um elemento.
- **Caso recursivo:** se o vetor ou subvetor possuir dois ou mais elementos, este deve ser dividido em dois subconjuntos a partir de sua posição central.

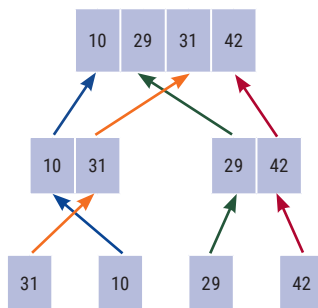
Ao analisar o processo recursivo anterior, pode-se perceber que este é o processo de divisão da técnica. Veja o vetor apresentado na divisão do merge sort, na figura 2.

Figura 2 – Processo de divisão do merge sort (as chamadas recursivas estão representadas pela cor azul, e os casos básicos, pela cor verde, como última etapa da divisão)



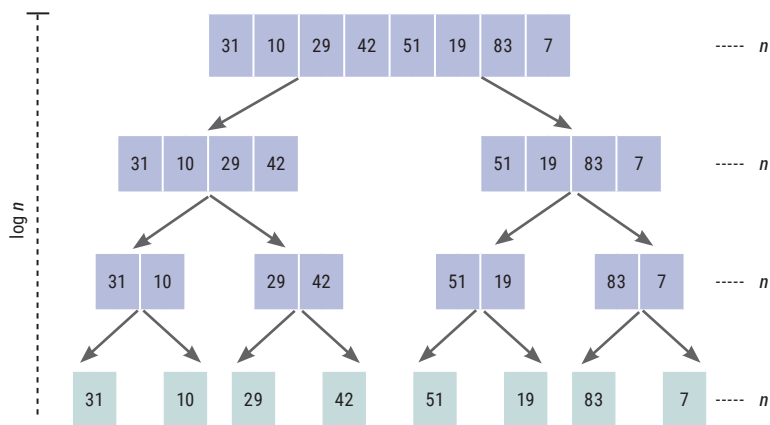
Após atingir o caso básico, o processo de conquista acontece e agora inicia-se o processo de combinação. Nesse momento, ocorre o retorno das funções recursivas para formar o vetor ordenado ao final. Cada subconjunto será concatenado de maneira a formar o subconjunto pai até que o problema original seja resolvido e o resultado final da ordenação seja obtido. A figura 3 apresenta dois níveis do algoritmo de retorno da função.

Figura 3 – Processo de retorno da chamada recursiva do merge sort, em que cada movimentação é representada por uma seta colorida, para cada nível do processo de ordenação (apenas dois níveis de ordenação foram apresentados para facilitar o entendimento)



Ao observarmos o processo do algoritmo merge sort, podemos imaginá-lo como uma árvore binária, a qual chamamos de árvore merge sort. Cada nó dessa árvore representa uma chamada recursiva. Os nós folhas, que seriam o último nível da árvore, são considerados os casos básicos do processo recursivo, como o nível com os nós de cor verde apresentado na figura 2, por exemplo. Neles, os subconjuntos estão totalmente ordenados e o retorno das chamadas recursivas inicia a ordenação durante a navegação aos nós pais, até que o vetor esteja totalmente ordenado, que é quando o nó raiz é atingido (GOODRICH; TAMASSIA, 2013). Com o conhecimento sobre a divisão em forma de árvore, a análise da eficiência desse algoritmo fica mais clara, acompanhe na figura 4.

Figura 4 – Análise do processo de execução do algoritmo merge sort, com base no exemplo da figura 1



Na figura 4, é utilizado o mesmo exemplo da figura 1, no entanto, a ênfase é exatamente na quantidade de operações que são realizadas pelo algoritmo. Agora, a altura da árvore obtida no processo recursivo executa o número de $\log n$ operações (log na base 2, normalmente omitido nas análises de algoritmos). Cada linha executa n operações. Ao combinar as duas funções, obtemos um total de $n \log n$ operações. Sendo assim, o merge sort pode ser considerado mais rápido que os algoritmos mais simples, como bubble sort, insertion sort e selection sort.

Nesse momento, após a compreensão do algoritmo, vamos realizar a sua implementação utilizando a linguagem de programação Java:

```
1. import java.util.Arrays;
2.
3. public class merge sort {
4.
5.     public static void sort(int X[], int inicio, int
fim) {
6.         if (inicio < fim) {
7.             int meio = (inicio + fim) / 2;
8.             sort(X, inicio, meio);
9.             sort(X, meio + 1, fim);
```

```

10.         merge(X, inicio, meio, fim);
11.         System.out.printf("Passo recursivo: %s \n",
Arrays.toString(X));
12.     }
13. }
14.
15.     private static void merge(int X[], int inicio, int
meio, int fim) {
16.         int i, esquerda, direita;
17.         int aux[] = new int[X.length];
18.
19.         for (i = inicio; i <= fim; i++) aux[i] = X[i];
20.
21.         esquerda = inicio;
22.         direita = meio + 1;
23.         i = inicio;
24.
25.         while (esquerda <= meio && direita <= fim) {
26.             if (aux[esquerda] <= aux[direita]) X[i++] =
aux[esquerda++];
27.             else X[i++] = aux[direita++];
28.         }
29.
30.         while (esquerda <= meio) X[i++] =
aux[esquerda++];
31.     }
32.
33.     public static void main(String[] args) {
34.         int X[] = {31, 10, 29, 42, 51, 19, 83, 7};
35.         sort(X, 0, X.length - 1);
36.         System.out.printf("Ordenado: %s \n", Arrays.
toString(X));
37.     }
38. }

```

O algoritmo é dividido em dois métodos, o método *sort* e o *merge*. No método *sort*, entre as linhas 5 e 13, ocorrem apenas as chamadas recursivas, dividindo o vetor ao meio enquanto a posição de início for menor que o fim. O método que exige mais capacidade de processamento é o *merge*. A parte mais onerosa da ordenação está entre as linhas 25 e 28,

nas quais as posições são comparadas e copiadas no local correto da ordenação.

Perceba que esse algoritmo apresenta uma implementação complexa e o uso da recursão é imprescindível para obter um melhor entendimento. Outro algoritmo em que ocorre essa situação é o quicksort, que será apresentado no próximo tópico.

2 Quicksort

Apesar de seu nome pressupor rapidez, em seu pior caso, o tempo de execução do algoritmo quicksort terá um número de operações igual à quantidade de elementos ao quadrado. Contudo, espera-se que seu tempo de execução médio seja $n \log n$, assim como o merge sort. Diferentemente deste, o processo mais custoso do quicksort em relação ao seu desempenho está na separação dos elementos para a chamada recursiva. Esse trabalho é realizado com base em um elemento de índice p , denominado de pivô. A determinação do elemento pivô pode ser realizada de maneira aleatória ou arbitrária (CORMEN *et al.*, 2002).

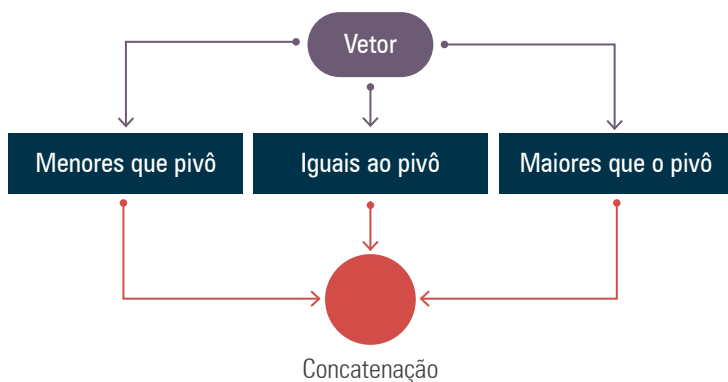
Apesar dessas considerações, o quicksort é um dos algoritmos mais empregados na tarefa de ordenação (GOODRICH; TAMASSIA, 2013). Seu desempenho é tão significativo que as soluções desenvolvidas em ambientes virtuais utilizam-no como algoritmo de ordenação (CORMEN *et al.*, 2002).

O processo de divisão e conquista do quicksort ocorre a partir do particionamento do vetor em três outros (GOODRICH; TAMASSIA, 2013):

- um vetor com os elementos menores que o pivô;
- um vetor com os elementos iguais ao pivô;
- um vetor com os elementos maiores que o pivô.

Veja a ilustração do processo de partição e concatenação do vetor em ordenação pelo algoritmo quicksort, na figura 5.

Figura 5 – Processo de partição do vetor original em três outros vetores, a partir de um elemento pivô, e a concatenação, já na posição ordenada

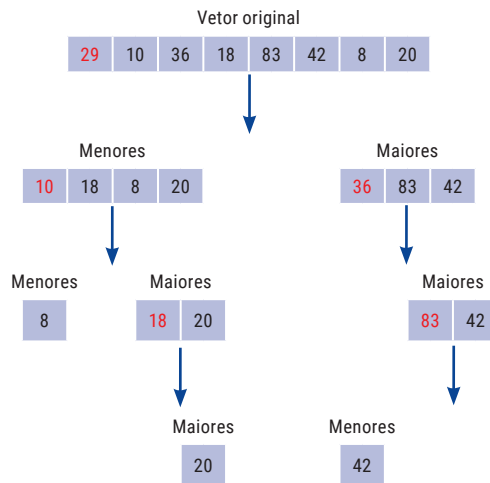


Fonte: adaptado de Goodrich e Tamassia (2013).

O pivô, como já observamos, pode ser obtido de maneira aleatória ou arbitrária, fixa, a partir das posições de início (0) ou fim (tamanho do vetor – 1). Não se recomenda utilizar uma posição ao centro do vetor, pois, em algum momento durante o processo de ordenação, essa posição pode não mais existir. Portanto, se não quiser utilizar as posições nas extremidades dos vetores, trabalhe com posições relativas (CORMEN et al., 2002).

A figura 6 apresenta o exemplo de separação do vetor a partir do particionamento com base em um pivô. Para o exemplo, o pivô será considerado como o primeiro elemento do vetor, o elemento armazenado na posição 0.

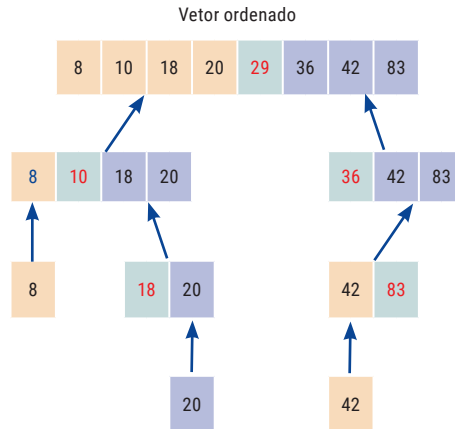
Figura 6 – Processo de particionamento adotando como pivô o primeiro elemento de cada conjunto (os pivôs estão sinalizados na cor vermelha)



O processo de partição é realizado até que os subconjuntos tenham o tamanho de um único elemento.

Após a separação, é necessário unir os subconjuntos para que retornem o vetor original de maneira ordenada. Como os vetores são separados por menores, iguais e maiores, a combinação entre eles é basicamente uma operação simples de junção já na sequência ordenada. A ordenação assim acontece de maneira automática (GOODRICH; TAMASSIA, 2013). A figura 7 apresenta a sequência de união dos vetores.

Figura 7 – Processo de união realizado pelo algoritmo quicksort após finalizar o particionamento do vetor (os quadrados em laranja representam os menores elementos, os verdes representam o pivô selecionado e, por fim, os azuis indicam os elementos maiores que o pivô)



Como no processo de particionamento eles foram totalmente separados, a união ocorre tranquilamente e pode-se observar que é um processo de concatenação simples entre os elementos menores, o pivô e os elementos maiores nessa sequência específica.

A implementação do algoritmo quicksort é apresentada a seguir na linguagem de programação Java:

```

1. import java.util.Arrays;
2.
3. public class quicksort {
4.
5.     public static void sort(int X[], int inicio, int
fim) {
6.         if (inicio < fim) {
7.             int pivot = divide(X, inicio, fim);
8.             sort(X, inicio, pivot - 1);
9.             sort(X, pivot + 1, fim);
10.        }
11.    }
12.
13.    public static int divide(int X[], int inicio, int
fim) {

```

```

14.         int pivot = X[inicio];
15.         int postPivot = inicio;
16.         for (int i = inicio + 1; i <= fim; i++) {
17.             if(X[i] < pivot) {
18.                 X[postPivot] = X[i];
19.                 X[i] = X[postPivot + 1];
20.                 postPivot++;
21.             }
22.         }
23.         X[postPivot] = pivot;
24.         return postPivot;
25.     }
26.
27.     public static void main(String[] args) {
28.         int X[] = {29, 10, 36, 18, 83, 42, 8, 20};
29.         sort(X, 0, X.length - 1);
30.         System.out.printf("Ordenado: %s \n", Arrays.
toString(X));
31.     }
32. }

```

Assim como no algoritmo merge sort, o quicksort é dividido entre dois métodos. No método *sort* ocorrem as chamadas recursivas, após a identificação do pivô (linhas 5 a 11). No método *divide*, ocorre a divisão, entre as linhas 13 e 25. O método *divide* posiciona o pivô ao centro do vetor e coloca nas posições anteriores a ele os menores elementos, enquanto os maiores são colocados nas posteriores. Este é um trabalho realizado pela referência de posição do vetor, o que facilita o uso de memória de armazenamento. O desempenho é bem próximo ao de se criar vetores novos, mas, ao criar vetores, utiliza-se mais espaço de memória.

A visualização da execução do quicksort é também como uma árvore binária. Entretanto, a altura da árvore no quicksort pode ser linear, em seu pior caso. O pior caso do algoritmo acontece quando ele é aplicado em um conjunto de dados totalmente distintos e já ordenado. Considerando que o pior caso é algo raro de ocorrer, o quicksort tem se mostrado a melhor opção quando o assunto é ordenação.

3 Exercícios de fixação

Para praticar, segue uma lista de exercícios.

1. Execute o algoritmo merge sort para os vetores abaixo e compare as diferenças entre os resultados:
 - a. [36, 15, 75, 2, 16, 48, 42, 51]
 - b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
 - c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
2. Execute o algoritmo quicksort para os vetores abaixo e compare as diferenças entre os resultados:
 - a. [36, 15, 75, 2, 16, 48, 42, 51]
 - b. [13, 45, 38, 20, 23, 7, 6, 5, 10, 11]
 - c. [9, 8, 7, 6, 5, 4, 3, 2, 1]
3. Compare o resultado entre os dois algoritmos.

Considerações finais

Os algoritmos demonstrados ao longo deste capítulo apresentam características mais sofisticadas para solucionar o problema da ordenação. Nos algoritmos apresentados, a técnica utilizada é a de dividir e conquistar, cuja intenção é dividir o problema complexo em pequenos subproblemas de resolução mais simples. Apesar de o cenário ser aplicado para a ordenação de dados, essa técnica pode ser empregada em qualquer cenário de soluções de problemas.

Quando se trata de ordenação, a técnica de divisão e conquista está amplamente vinculada ao processo de algoritmos recursivos, que nessa situação é a melhor opção para diminuir a complexidade da solução e deixar o código mais compreensível. Os dois algoritmos apresentados,

merge sort e quicksort, são considerados os algoritmos mais rápidos para ordenação, de modo geral. Outros algoritmos possuem tempos de execução mais rápidos, como o radix sort, mas limitam-se a alguns cenários, como o de dicionário de dados.

Referências

CORMEN, Thomas H. *et al.* **Algoritmos**: teoria e prática. Rio de Janeiro: Editora Campus, 2002.

FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Prentice Hall, 2005.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

SEDGEWICK, Robert; FLAJOLET, Philippe. **An introduction to the analysis of algorithms**. [S. l.]: Pearson Education India, 2013.

Eficiência de algoritmos

Quando é preciso encontrar a solução para algum problema, geralmente se constrói um algoritmo, e um algoritmo nada mais é do que o passo a passo que conduz o caminho para a solução. Entretanto, um problema pode conter diversas soluções (NECAISE, 2010). Com diversas possibilidades para solucionar um determinado problema, é necessário descobrir quais as características que podem classificar e avaliar se um determinado algoritmo pode transitar entre diversas aplicações, ou mesmo comparar quais algoritmos são melhores que outros dentro de um determinado cenário (CORMEN *et al.*, 2009; NECAISE, 2010).

Algumas das principais características que guiam os estudos com algoritmos estão relacionadas a recursos computacionais como tempo de processamento e espaço de memória ocupado. Em geral, o foco das análises sempre gira em torno do tempo de processamento de um algoritmo comparado a outro dentro de um mesmo contexto (CORMEN *et al.*, 2009).

A análise do tempo de processamento de um algoritmo depende de alguns fatores importantes (NECAISE, 2010), como:

- A quantidade de informação que irá compor a entrada do algoritmo influencia diretamente a variável do tempo de processamento.
- O hardware que é utilizado na comparação também influencia o resultado do desempenho computacional do algoritmo. Condições como a realização de tarefas simultâneas e até mesmo a hora da execução pode proporcionar tendências nas comparações.
- Outra influência é a linguagem de programação escolhida para implementar os algoritmos, pois os resultados obtidos com uma linguagem totalmente compilada são diferentes dos obtidos com uma pré-compilada ou interpretada. Além disso, existem linguagens que possuem otimizações para certos cenários, transparentes ao desenvolvedor.

Outros interesses podem surgir ao comparar algoritmos, como consumo eficiente da bateria em dispositivos móveis; a acurácia do resultado obtido ao final do processamento das informações de um determinado algoritmo; ou até mesmo o consumo de memória em microprocessadores, devido às suas limitações ou por ser um dispositivo embarcado (CORMEN *et al.*, 2009).

Ao construir um algoritmo, é importante saber que, geralmente, ele não é capaz de ter um tempo de processamento bom e de ocupar uma quantidade baixa de memória. Em linhas gerais, pode-se dizer que

quanto mais espaço um algoritmo ocupar na memória, mais rápido será seu tempo de execução dado sua entrada. Isso quer dizer que o crescimento da ocupação de memória está ligado diretamente com o menor tempo de processamento do algoritmo (CORMEN *et al.*, 2009).

Sendo assim, a análise de algoritmos tem como objetivo prover ferramentas para que os desenvolvedores sejam capazes de tomar decisões sobre qual o melhor algoritmo para o problema que deve ser solucionado. A ferramenta que encabeça toda a base no âmbito da análise de algoritmos é a matemática. Ela permite que os estudos possam ir de análises em linhas gerais, quando o problema é complexo, até resultados analíticos específicos a cada problema (CORMEN *et al.*, 2009).

Ao longo deste livro, o foco da comparação dos algoritmos terá como base a relação do tempo de processamento para a conclusão de uma tarefa. Nenhum cenário específico será utilizado para não gerar uma tendência na aplicação da análise de algoritmos.

1 Matemática como ferramenta para análise de algoritmos

Existem diversos casos em que a fronteira entre a matemática e a computação se estreita, desde a fabricação de hardwares até a construção de softwares. Na construção de softwares, esse estreitamento está relacionado tanto ao que existe por trás do código, a linguagem de máquina, como à lógica, e também representa as principais funções de comparação entre os algoritmos e suas definições formais.

Ao todo, pode-se dizer que existem sete funções comumente encontradas em toda a literatura de análise de algoritmos. Verifica-se cada uma delas e ao final um comparativo é feito sobre quais são as melhores e qual seria o alvo para buscar o resultado ótimo de acordo com o cenário de aplicação (GOODRICH; TAMASSIA, 2013).

1.1 Função constante

Com absoluta certeza, a função mais simples dentre todas as sete comuns é a função constante. Sua definição formal é dada através da equação:

$$f(n) = c$$

Em que c é um número fixo, não se alterando ao longo das variações de n .

Como o objetivo da análise é dado por funções que retornam um número inteiro, a função constante mais utilizada nas análises, nesse caso, é $f(n) = 1$ (GOODRICH; TAMASSIA, 2013). Além disso, a função constante pode ser relacionada a uma segunda função, como se fosse um peso para o resultado dela. Veja a representação formal:

$$f(n) = c \cdot g(n)$$

A função constante, então, costuma representar a quantidade de passos que um algoritmo executa em uma determinada aplicação.

1.2 Função logarítmica

Entre as sete funções, a que se demonstra mais onipresente é a função logarítmica (GOODRICH; TAMASSIA, 2013). Sua definição formal é dada pela seguinte equação:

$$f(n) = \log_b n$$

Para qualquer constante $b > 1$, sendo b definido como a base do logaritmo.

Uma regra importante para essa função é que, por definição, o $\log_b 1 = 0$. É possível efetuar uma aproximação efetiva para uma função de logaritmo. A função logarítmica é a quantidade de vezes que se pode dividir o n por b até chegar ao valor 1 ou próximo. Observe o exemplo elaborado por Goodrich e Tamassia (2013):

$$\log_3 27 \rightarrow 27 / 3 / 3 / 3 = 1$$

O resultado da função logarítmica de 27 na base 3 é igual a 3, uma vez que, para alcançar o valor 1, foram necessárias 3 divisões de 27 por 3. Veja outro exemplo:

$$\log_2 12 \rightarrow 12 / 2 / 2 / 2 / 2 = 0,75 \leq 1$$

Nesse caso, a divisão ficou abaixo de 1 e não exatamente 1. Por aproximação, considera-se que o resultado da função logarítmica de 12 na base 2 é igual a 4. Na base 2, aceita-se a aproximação, uma vez que diversos algoritmos utilizam a estratégia de divisão e conquista, quebrando sempre entre dois subproblemas o problema original (GOODRICH; TAMASSIA, 2013). Como a base 2 é a mais comum ao fazer uma análise de algoritmo, toda vez que é mencionada a função $\log n$ subentende-se que se trata de um log na base 2.



IMPORTANTE

O log é a quantidade de vezes que se pode dividir um determinado número pela base até que o resultado das divisões subsequentes seja próximo ou igual a 1.

Com esse pequeno truque matemático, é possível chegar ao valor de um logaritmo sem a necessidade de nenhum cálculo complexo.

1.3 Função linear

Assim como a função constante, a função linear também é uma função simples e igualmente importante no processo de análise de algoritmos (GOODRICH; TAMASSIA, 2013). Sua definição formal é dada pela seguinte equação:

$$f(n) = n$$

Isso significa que toda entrada n informada à função resultará em uma saída igual à entrada. Essa função é facilmente encontrada em diversos cenários da computação. Toda vez que é executada uma operação para n elementos, por exemplo, no caso de operações que percorrem um vetor por completo, a função que descreve essa operação é justamente a função linear (GOODRICH; TAMASSIA, 2013).

1.4 Função $n \log n$

A função $n \log n$ é uma função que possui uma taxa de crescimento igual ao valor de sua entrada multiplicado pelo valor de $\log n$ na base 2, conforme a definição formal (GOODRICH; TAMASSIA, 2013):

$$f(n) = n \cdot \log n$$

Ela possui uma taxa de crescimento superior à função linear, mas é bem menor quando comparada a funções polinomiais de grau 2 ou mais. A função $n \log n$ é o alvo para pesquisadores que querem otimizar um problema de ordem quadrática (GOODRICH; TAMASSIA, 2013).

1.5 Função quadrática

Quando a entrada n de uma função f atribui à sua saída o produto de n pelo próprio n , obtém-se uma função quadrática. Confira sua definição formal na equação a seguir:

$$f(n) = n^2$$

Isso ocorre, dentro do cenário da computação, geralmente em algoritmos que possuem laços de repetição aninhados. Em outras palavras, isso quer dizer que existe um laço de repetição externo que executa n vezes, linearmente, e um outro laço de repetição interno que também executa linearmente n vezes (GOODRICH; TAMASSIA, 2013).

Apesar de ser uma função polinomial, assim como a linear, a função quadrática é recorrente dentro do processo de análise de algoritmos.

1.6 Função cúbica e demais polinomiais

Assim como a função quadrática, a função cúbica atribui o produto de uma entrada n pelo próprio n , só que, nesse caso, três vezes. A função cúbica é menos comum no cenário de análise de algoritmos. Sua definição formal está representada na equação a seguir:

$$f(n) = n^3$$

Contudo, durante a análise de algoritmos, outras funções polinomiais, com graus maiores que a linear, a quadrática e a cúbica, podem ser encontradas. De modo geral, um polinômio é definido formalmente da seguinte maneira:

$$f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \dots + a_dn^d$$

Em que:

- a_0, a_1, \dots, a_d são os coeficientes do polinômio (as constantes) e $a_d \neq 0$.
- O inteiro d indica a maior potência de um termo no polinômio. Ele também representa o grau do polinômio.

Com base nessa definição, pode-se unificar as funções linear, quadrática e cúbica como funções polinomiais. Porém, devido ao contexto da aplicação – a análise de algoritmos –, é conveniente mantê-las separadas, tendo em vista que seu uso é mais comum assim (GOODRICH; TAMASSIA, 2013). Funções polinomiais com grau maior ou igual a 3 devem ser evitadas. Quando são encontradas em algoritmos, devem-se realizar estudos para que este seja otimizado.

1.7 Função exponencial

A última função mais comumente encontrada em análise de algoritmos é a função exponencial. Sua definição formal é apresentada pela equação a seguir:

$$f(n) = b^n$$

Em que:

- b é uma constante positiva, chamada de base; e
- n é o expoente recebido como argumento da função.

O resultado de $f(n)$ é obtido através da multiplicação da base por ela mesma, o número de vezes informado pela entrada n . Na análise de algoritmos, a base 2 é tida como a mais comum. Portanto, em geral, quando se fala de função expoente ou exponencial, subentende-se que se trata da função $f(n) = 2^n$ (GOODRICH; TAMASSIA, 2013).

2 Comparativo das taxas de crescimento

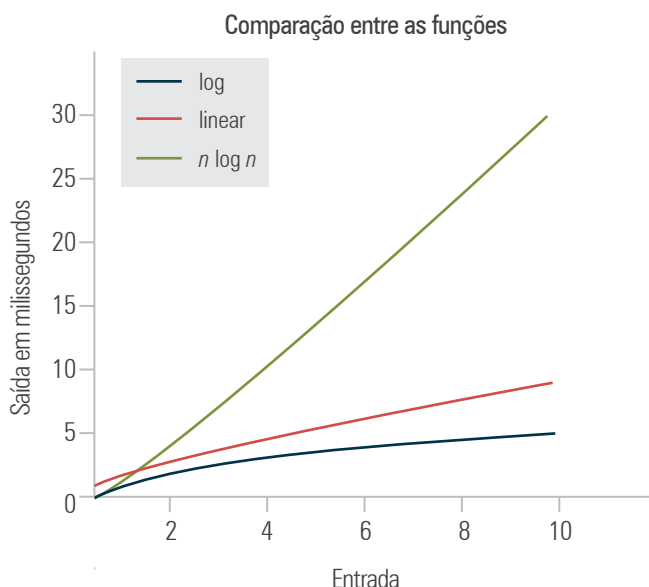
As funções podem ser organizadas dentro de uma tabela que demonstra a sequência hierárquica de uma função de acordo com o tempo de processamento, indo do menor tempo até o mais demorado entre elas. O quadro 1 apresenta a sequência hierárquica mencionada, sendo a coluna da extrema esquerda a função mais rápida e a da extrema direita, a mais demorada.

Quadro 1 – Tabela representando as sete funções para a análise de algoritmos, sendo da mais rápida até a mais demorada, da esquerda para a direita

CONSTANTE	LOGARÍTMICA	LINEAR	$N \log N$	QUADRÁTICA	CÚBICA	EXPONENCIAL
1	$\log n$	n	$n \log n$	n^2	n^3	2^n

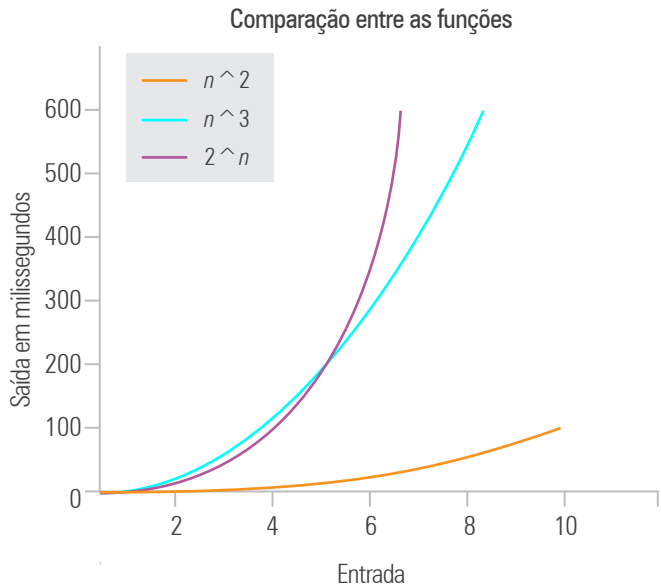
Essas taxas de crescimento podem ser utilizadas para definir as métricas de atuação, dependendo da aplicação. Por exemplo, quando se trata de estrutura de dados, existe um desejo de que as operações executadas sejam todas entre as funções constante e logarítmica. Já para grande parte dos algoritmos, o desejo é por um processamento linear ou $n \log n$. As funções polinomiais de grau 2 e 3 ainda são toleradas em casos muito específicos, mas não são desejadas, nem são alvo de qualquer desenvolvedor de algoritmos. Já a função exponencial é, computacionalmente, impraticável. Soluções exponenciais podem ser aceitas para entradas muito pequenas, mas, no geral, devem ser evitadas e, quando ocorrem, estudos de como otimizá-las devem ser conduzidos (GOODRICH; TAMASSIA, 2013). Os gráficos 1, 2 e 3 apresentam a comparação entre as taxas de crescimento. O gráfico 1 apresenta as funções logarítmica, linear e $n \log n$. A função constante foi omitida por sua representação ser uma linha constante na horizontal. O gráfico 2 tem as funções quadrática, cúbica e exponencial. O gráfico 3 apresenta uma comparação entre elas.

Gráfico 1 – Apresentação gráfica das taxas de crescimento das funções logarítmica, linear e $n \log n$ para comparação visual



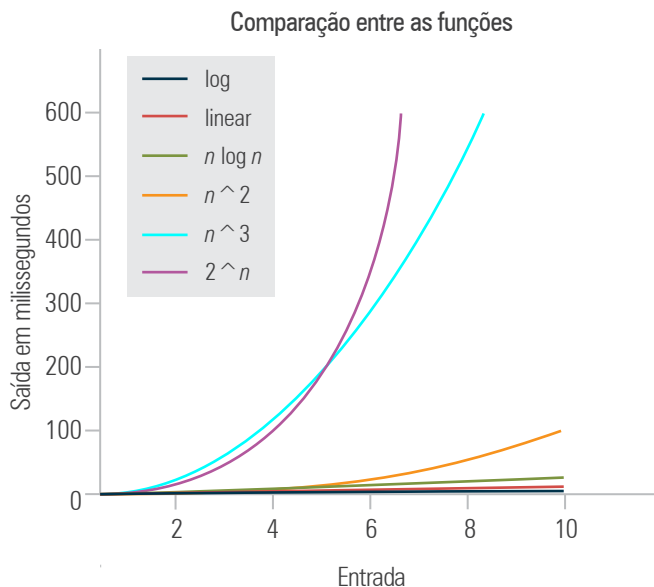
Veja no gráfico 1 como a taxa de crescimento logarítmica (linha azul) se mantém bem abaixo da função linear (linha vermelha). Para uma entrada igual a ou menor que 2, a função $n \log n$ (linha verde) também se mantém com uma taxa de crescimento menor que a linear. A partir de uma entrada n maior que 2 o crescimento já começa a aumentar. Para efeito comparativo, com uma entrada igual a 10, a função logarítmica possui uma saída de aproximadamente 2 milissegundos; a função linear atinge o processamento de 10 milissegundos; e a função $n \log n$ está com o processamento em torno de 35 milissegundos. Obviamente, milissegundo não é uma medida tão significativa com a qual se preocupar em relação ao desempenho do algoritmo. Mas, em um cenário escalável, o tempo de processamento começa a influenciar. A comparação continua com o gráfico 2 para as demais funções.

Gráfico 2 – Apresentação gráfica das taxas de crescimento das funções quadrática, cúbica e exponencial para comparação visual



A função quadrática (linha laranja) parece até uma boa escolha quando comparada com as funções cúbicas (linha azul) e exponencial (linha roxa). Porém, ao observar sua saída para a entrada de 10 elementos, ela levou 100 milissegundos para chegar ao resultado. Para uma melhor visualização do comportamento das funções, o gráfico ficou limitado à saída de 600 milissegundos, mas a função cúbica alcança o processamento de 512 milissegundos já com 8 elementos de entrada. A função exponencial com 10 elementos de entrada supera os 20.000 milissegundos de processamento. Note que, se um algoritmo for escalado para uma entrada de 1.000 registros, essas funções ficam impraticáveis. Por isso o desejo de sempre manter um algoritmo no máximo com a função $n \log n$. O gráfico 3 apresenta todas as funções para uma melhor visualização dessa justificativa.

Gráfico 3 – Apresentação da comparação entre todas as funções para análise de algoritmos



No gráfico 3, é possível notar que a função logarítmica quase desaparece na escala com a função exponencial, mesmo limitada a 600 milissegundos. Com esse gráfico, nota-se o comportamento de um algoritmo de acordo com a sua implementação. Por isso, a construção de qualquer algoritmo deve ser planejada e pensada com calma.

Outro ponto importante é que, ao fazer uma análise de algoritmos, o intuito é determinar qual deles é o melhor. Como essa comparação geralmente é feita levando em consideração o tempo de processamento deles, a variável da taxa de crescimento da função passa a ser um fator de qualidade do algoritmo (GOODRICH; TAMASSIA, 2013).

3 Identificando a função do algoritmo

Como discutido nos tópicos anteriores, uma das maneiras de obter o tempo de processamento é realizando a medida ao executar o

algoritmo. Mas existem diversas afirmações sobre por que mensurar com base na execução não é uma boa ideia. Sendo assim, como encontrar a função que representa o algoritmo, de um modo que não dependa do equipamento e que seja a mais fidedigna possível? (CORMEN *et al.*, 2009; NECAISE, 2010).

Para solucionar esse problema, é recomendável criar medidas mensuráveis e verificar que partes do algoritmo afetam diretamente o tempo de processamento das entradas. Assim, é possível identificar a ordem de magnitude daquele algoritmo, em outras palavras, a função real que representa as operações por ele realizadas (NECAISE, 2010).

O primeiro passo é estipular o problema, implementar o algoritmo e, na sequência, fazer a sua análise, linha a linha. Como exemplo, é realizada a análise de uma algoritmo que implementa um somatório de uma matriz 3 por 5. Perceba que apesar de a análise ser feita em uma matriz com uma determinada dimensão, seu resultado deve ser genérico o suficiente para ser aplicado para quaisquer valores de entrada. A classe SomatorioMatriz é apresentada a seguir:

```
1. public class SomatorioMatriz {
2.
3.     public static void main(String[] args) {
4.
5.         int[][] matriz = {
6.             {1, 2, 3, 4, 5},
7.             {6, 7, 8, 9, 10},
8.             {11, 12, 13, 14, 15}
9.         };
10.
11.         int[] somaDasLinhas = {0, 0, 0};
12.         int totalDaSoma = 0;
13.
14.         for(int i = 0; i < matriz.length; i++) {
15.             for(int j = 0; j < matriz[i].length;
16.                 j++) {
```

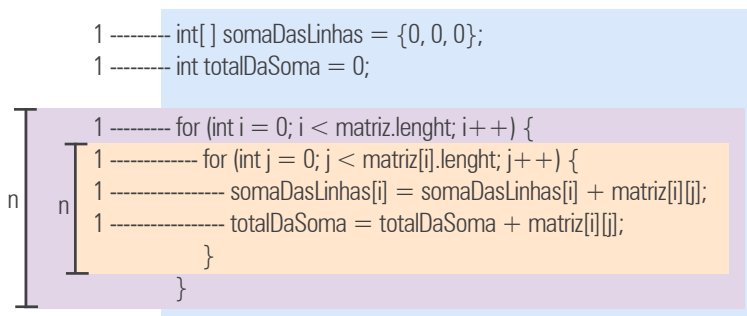
```

16.                somaDasLinhas[i] =
somaDasLinhas[i] + matriz[i][j];
17.                totalDaSoma = totalDaSoma +
matriz[i][j];
18.            }
19.        }
20.
21.        System.out.printf("O soma total eh: %d\n",
totalDaSoma);
22.    }
23. }

```

Na classe *SomatorioMatriz*, o algoritmo está definido dentro do método *main*, realizando desde as declarações das variáveis auxiliares até o processamento do resultado final, entre as linhas 11 e 19. Da linha 5 até a 9, é declarada a matriz de entrada para o algoritmo. A figura 1 apresenta o trecho de código entre as linhas 11 e 19, com destaque para entender melhor o processo de análise do algoritmo.

Figura 1 – Destaque do valor de cada operação e bloco de instrução na análise do algoritmo da somatória de matrizes



Na figura 1, é apresentada a análise do algoritmo para efetuar a soma entre todos os elementos de uma matriz. Observe, na parte azul, que existe a declaração de duas variáveis, independentemente do espaço de memória que vão ocupar, cada uma dessas operações é executada em um tempo constante 1. Na sequência, temos o primeiro *for* (bloco roxo).

A linha do *for*, especificamente, leva um tempo constante 1 para ser executada, porém essa linha é executada n vezes, sendo a quantidade de linhas da matriz. Dentro desse *for*, existe um outro *for* (bloco laranja), que também tem o mesmo padrão de execução. Contudo, o segundo *for* está aninhado ao primeiro, portanto ele executa n vezes multiplicadas por n . Essa análise é feita pois, para cada iteração do *for* externo, ele executará seu bloco n vezes.

Por fim, dentro do segundo *for*, existem duas operações sendo executadas a um tempo constante 1 cada. Juntando as operações de cada bloco e linha, temos uma função n^2 que executa duas operações constantes. A equação dos blocos *for* é definida como $2n^2$. Somando o tempo constante das duas operações isoladas no bloco azul, a equação desse algoritmo fica:

$$2n^2 + 2$$

Agora, pode-se dizer que o algoritmo de somatório de matrizes tem uma função $f(n) = 2n^2 + 2$. Por uma questão de simplificação, é dito, em termos gerais, que esse é um algoritmo com função n^2 . Omitem-se as constantes, com o intuito de simplificar a análise. Apenas em algoritmos de mesma grandeza que esse tipo de detalhe é levado em consideração, para que uma classificação mais precisa seja determinada (CORMEN *et al.*, 2009; NECAISE, 2010).

4 Notação Big-O

Em geral, como visto ao longo do capítulo, a classificação do algoritmo é realizada com base na sua ordem de magnitude, ou seja, a função que o representa. A preferência pela ordem de magnitude como medida de classificação ocorre porque ela pode ser utilizada tanto para o tempo de processamento como para a ocupação de espaço na memória (DEITEL; DEITEL, 2008).

A análise feita para atribuir a ordem de magnitude do algoritmo é chamada de análise assintótica. Para facilitar a classificação de um algoritmo em um determinado ranking, cunhou-se um termo específico e bem comum em qualquer literatura: notação Big-O. Em análise de algoritmos, devemos ler a notação Big-O como “em ordem de” (HARDY; LITTLEWOOD, 1914; KNUTH, 1976). A função $f(n)$ determina a taxa de crescimento de um algoritmo baseando-se no aumento do número de elementos que compõem a entrada n (CORMEN *et al.*, 2009; NECAISE, 2010).

A complexidade de um algoritmo é determinada com base na função de crescimento. Então, após a conclusão da análise, é dito que o algoritmo executa suas operações na ordem de $f(n)$. A notação formal para essa expressão é dada por $O(f(n))$ (NECAISE, 2010).

Quando uma análise assintótica é realizada em um algoritmo, o objetivo principal é encontrar a função que determine o limite superior do tempo de processamento do algoritmo. O limite superior também é conhecido como o cenário de pior caso de um algoritmo. Pense que, se o algoritmo tiver um bom desempenho dentro do pior cenário possível, ele irá conseguir processar as informações dos cenários menos complexos de uma maneira muito mais tranquila (CORMEN *et al.*, 2009; NECAISE, 2010).

A notação Big-O representa justamente o limite superior, baseando-se no grau mais alto encontrado para n (DEITEL; DEITEL, 2008). Dois algoritmos podem ter a mesma taxa de crescimento baseando-se na função $f(n)$, porém, em uma análise mais detalhada, descobre-se que existe um valor constante que irá determinar uma classificação melhor ou pior. Essa constante apresentada como fator determinante para definir qual o melhor algoritmo é chamada de constante da proporcionalidade (NECAISE, 2010).

Para auxiliar o processo de análise, algumas considerações devem ser mencionadas. Considere que todas as operações (linhas de código) entram na avaliação de um algoritmo, seja uma simples condição lógica ou operações aritméticas. Nesse ponto, assume-se que quaisquer operações básicas ou declarações levam o mesmo tempo de processamento.

Em um nível mais abstrato, segundo a terminologia, ele passa a ser chamado de tempo constante (DEITEL; DEITEL, 2008; NECAISE, 2010).

Os passos do algoritmo que possuem o tempo constante, geralmente, são omitidos, porque são representados através da declaração da constante da proporcionalidade inserida dentro da definição da equação. A partir desse momento, a classificação de um algoritmo é realizada pelo termo dominante da função. Esse termo é tão grande e com valores de entradas n tão altos que os demais termos da função podem ser ignorados. Por exemplo, com a equação $2n^2 + 15n + 500$, o termo n^2 torna-se dominante na equação inteira, fazendo com que os demais termos tenham uma relevância muito baixa e sejam insignificantes ao resultado final (CORMEN *et al.*, 2009; NECAISE, 2010). Daí em diante, toda a classificação dos algoritmos é realizada através da notação Big-O (NECAISE, 2010).



PARA PENSAR

Comumente, operações realizadas com a classe String possuem o tempo de processamento de acordo com o seu tamanho. Contudo, durante a análise, assume-se que a String possui um valor constante na operação.

5 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Qual função representa o algoritmo abaixo?

```
// n é informado pelo usuário
n = ?
cont = 0
i = 1
while(i <= n) {
    print i
}
```

```

        i = i + 1
        cont = cont + 1
    }
    print cont

```

2. Qual função representa o algoritmo abaixo?

```

// n é informado pelo usuário
n = ?
cont = 0
i = 1
while(i <= n) {
    print i
    i = i * 2
    cont = cont + 1
}
print cont

```

3. Qual função representa o algoritmo abaixo?

```

// n é informado pelo usuário
i = 1
j = 1
while(i <= n) {
    i = i * 2
}

while(j <= n) {
    j = j + 1
}

```

Considerações finais

Neste capítulo, foram apresentadas as diversas maneiras pelas quais um algoritmo pode ser avaliado e classificado. Além disso, apresentamos as sete funções mais comuns para análise de algoritmos, as quais devem ser aplicadas no dia a dia. Cada função apresentada na análise dos algoritmos é utilizada na notação Big-O para saber qual algoritmo é melhor para cada situação.

Variáveis como consumo de bateria, ciclo de processamento, velocidade e armazenamento podem ser fatores de classificação do algoritmo, utilizando a notação Big-O. E foram apresentadas também quais são as funções em cada tipo de aplicação de algoritmos e estrutura de dados.

Referências

CORMEN, Thomas H. *et al.* **Introduction to algorithms**. Cambridge: MIT Press, 2009.

DEITEL, Paul J.; DEITEL, Harvey M. **Java: como programar**. São Paulo: Pearson, 2008.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

HARDY, G. H.; LITTLEWOOD, J. E. Some problems of Diophantine approximation. **Acta Mathematica**, v. 37, n. 1, 1914.

KNUTH, Donald. Big Omicron and big Omega and big Theta. **SIGACT News**, apr./june 1976.

NECAISE, Rance D. **Data structures and algorithms using Python**. Hoboken: John Wiley & Sons, Inc., 2010.

XAVIER, Gley F. C. **Lógica de programação**. São Paulo: Editora Senac São Paulo, 2014.

Filas

Após realizar suas compras em um supermercado, um cliente toma a direção dos caixas para realizar o pagamento. Ao se aproximar dos guichês, encontra um conjunto de pessoas dispostas em uma linha. Com o passar do tempo, mais e mais pessoas chegam e se posicionam ao final dessa linha, aguardando sua vez de serem atendidas. Esse é um cenário cotidiano que ocorre em diversos lugares, desde um simples supermercado até um atendimento hospitalar. A linha de pessoas define um termo comumente chamado de fila.

A fila é uma estrutura muito presente também em aplicações de computadores. Ela é utilizada sempre que existe a necessidade de processar as informações na sequência em que elas chegam ao sistema. Algumas das aplicações que usufruem da estrutura de dados em formato de fila são: simulações computacionais, o escalonador de tarefas da CPU, o gerenciamento de impressões, o sistema de reservas na venda de passagem aéreas de uma companhia, entre outras (NECAISE, 2010; GOODRICH; TAMASSIA, 2013).

Definindo-a de maneira formal, a fila é uma lista especializada com um número limitado de operações, sendo que um novo elemento só pode ser adicionado em seu final e apenas o primeiro elemento dela pode ser removido. Por essa limitação em suas operações, a fila é conhecida com uma estrutura de dados FIFO (*first-in first-out*, em inglês), ou seja, o primeiro elemento que entra é o primeiro a sair, em uma tradução livre desse conceito. A figura 1 apresenta uma ilustração da estrutura de dados em fila (CORMEN *et al.*, 2009; NECAISE, 2010; GOODRICH; TAMASSIA, 2013).

Figura 1 – Representação da operação FIFO, na qual o elemento 8 é adicionado ao final da fila e o elemento 29 é removido em seu início



Na figura 1, é possível acompanhar as operações existentes na fila. Perceba que em seu início o número 29 é removido, a transição é representada pelo elemento com um tom mais claro na posição antes da operação e o elemento mais opaco na posição atual, ou após a operação. A partir da saída do número 29, o próximo número que deve sair da fila é o número 10, que assume a primeira posição da fila. No final da fila, o elemento 8 é inserido. Perceba que o elemento posicionado fora da fila também se apresenta mais claro, demonstrando sua posição

anterior ao adicioná-lo. Dentro da fila, ele fica mais opaco, determinando sua posição ao final da operação de adicionar o elemento na fila.

A definição de uma estrutura do tipo fila deve considerar os métodos básicos, que determinam as operações que serão realizadas por ela (DEITEL; DEITEL, 2008; NECAISE, 2010). Os métodos que devem ser considerados são:

- **ehVazia (isEmpty, em inglês):** informa um resultado booleano dizendo se a fila contém algum elemento.
- **tamanho (length, em inglês):** retorna a quantidade de elementos armazenados dentro da fila num determinado momento.
- **primeiro (front, em inglês):** retorna o elemento que está na posição frontal da fila, porém este é um método de consulta, não remove o elemento.
- **enfileira (enqueue, em inglês):** insere um elemento ao final da fila. Esse método deve receber como parâmetro o elemento que é inserido na fila.
- **desenfileira (dequeue, em inglês):** remove e retorna o elemento que está no início da fila. Uma regra importante nesse método é que nenhum elemento pode ser removido de uma lista vazia, portanto uma mensagem de erro deve retornar caso isso ocorra (GOODRICH; TAMASSIA, 2013).

Agora que os métodos básicos foram apresentados, vamos para as implementações de uma fila. A base para armazenar os dados é uma lista, com operações específicas. Portanto, é possível implementar uma fila utilizando diversos tipos de listas, como um vetor estático, um vetor dinâmico, uma lista circular ou uma lista ligada. Cada implementação será discutida e apresentada nas próximas seções.

1 Implementando a fila com vetores

Em Java, existe a possibilidade de se trabalhar com um vetor primitivo ou um vetor fornecido pela API Collections, implementada entre seus pacotes. A diferença entre os dois basicamente é que, na versão primitiva, seu tamanho é predefinido e seus métodos devem ser todos implementados. Utilizando a API Collections, a coleção utilizada como base possui os métodos de suporte implementados e seu tamanho é dinâmico, podendo armazenar a quantidade de elementos de acordo com a necessidade da aplicação e a capacidade do hardware.



PARA SABER MAIS

A API Collections do Java possui uma infinidade de coleções para o trabalhos com as estruturas de dados existentes na computação. É uma API muito importante e deve ser estudada em detalhes. Para conhecê-la, acesse o link: <https://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html> (acesso em: 20 abr. 2020).

Para compreender melhor o funcionamento de cada um dos métodos da fila, a primeira implementação será realizada com base no vetor primitivo. A classe Fila implementada deve receber em seu construtor a capacidade máxima que a fila irá armazenar. Além disso, os cinco métodos básicos também devem ser implementados para que a classe fique completa. Alguns atributos são declarados para que sirvam de suporte aos controles de manipulação dos dados.

Como a implementação está sendo realizada com o vetor primitivo, é necessário também efetuar a validação do vetor com a capacidade máxima. Assim, o ciclo fica completo, e a classe, pronta para servir a qualquer aplicação. Confira a seguir a implementação da classe Fila utilizando o vetor primitivo.

```

1.  public class Fila {
2.      private final int[] dados;
3.      private final int capacidade;
4.      private int fim;
5.
6.      public Fila(int capacidade) {
7.          this.capacidade = capacidade;
8.          this.dados = new int[this.capacidade];
9.          this.fim = 0;
10.     }
11.
12.     public Fila() {
13.         this(10);
14.     }
15.
16.     public int primeiro() {
17.         return this.dados[0];
18.     }
19.
20.     public boolean ehVazia() {
21.         return this.fim == 0;
22.     }
23.
24.     public int tamanho() {
25.         return this.fim;
26.     }
27.
28.     public void enqueue(int numero) {
29.         if((this.fim + 1) > this.capacidade) {
30.             throw new RuntimeException("A fila
está com a capacidade máxima.");
31.         }
32.         this.dados[this.fim++] = numero;
33.     }
34.
35.     public int dequeue() {
36.         if(this.ehVazia()) {
37.             throw new RuntimeException("A fila
está vazia.");
38.         }
39.         int numero = this.dados[0];

```

```

40.         for(int i = 0; i < this.fim - 1; i++) {
41.             this.dados[i] = this.dados[i + 1];
42.         }
43.         this.fim--;
44.         return numero;
45.     }
46. }

```

Uma classe *Fila* é criada para representar a estrutura de dados. Como a implementação é realizada com um vetor do tipo primitivo, são necessários três atributos: um vetor do tipo *int* (para facilitar a implementação e compreensão) chamado *dados* (linha 2); um inteiro chamado *capacidade* (linha 3), que representa a capacidade máxima de armazenamento da fila; e um inteiro chamado *fim* (linha 4), que representa a próxima posição disponível para armazenar um elemento. Quando o atributo *fim* for igual a 0 significa que a fila está vazia.

Na sequência, dois construtores da classe são criados. O primeiro, entre as linhas 6 e 10, recebe como parâmetro a capacidade de armazenamento desejada para a fila. Nele são inicializados os atributos da capacidade, que é utilizada para criar a instância do vetor e o atributo *fim* iniciado com o valor de 0. O segundo construtor, declarado entre as linhas 12 e 14, não recebe nenhum parâmetro. Ele apenas invoca o primeiro construtor, determinando a capacidade máxima dele para 10 elementos. Em seguida, os três métodos para suporte são declarados: o método *primeiro* (linhas 16 a 18), que retorna o elemento armazenado na primeira posição do vetor sem removê-lo; o método *ehVazia* (linhas 20 a 22), que retorna o valor booleano da comparação entre a igualdade do atributo *fim* e o valor 0 (quando verdadeira, a comparação significa que a fila está vazia); e o método *tamanho* (linhas 24 a 26), que retorna a quantidade de elementos armazenados na fila, representada pelo atributo *fim*.

Logo após a definição dos métodos que dão suporte à estrutura fila, são estabelecidos os métodos *enfileira* (linhas 28 a 33) e o método

desenfileira (linhas 35 a 45). O método *enfileira* necessita verificar se a fila está cheia antes de inserir um novo elemento. Para essa comparação, verifica-se se a posição *fim* + 1 é maior que a capacidade da fila. Caso seja verdade, uma exceção é lançada (linha 30), com a mensagem "A fila está com a capacidade máxima.", impedindo a operação de ser realizada. No caso de existir espaço, o elemento recebido no parâmetro é inserido na posição *fim* e, na sequência, *fim* é incrementado em 1. O método *desenfileira* tem a obrigatoriedade de verificar se a fila está vazia.

Para isso, ele utiliza o método *ehVazia* na condição. Caso o método retorne positivo para a verificação da fila vazia, uma exceção é lançada com a seguinte mensagem "A fila está vazia." (linha 37). Caso exista algum elemento na fila, o processo de remoção é iniciado. O número a ser removido é armazenado em uma variável auxiliar (linha 39). Entre as linhas 40 e 42 é implementado um laço de repetição do tipo *for* para reposicionar todos os elementos da fila a partir da primeira posição, respeitando a ordem de entrada. Na linha 43, é realizado o decremento do atributo *fim*. Por fim, o valor removido é retornado (linha 44).

A classe App apresentada a seguir faz o teste da classe Fila, em uma fila de capacidade máxima igual a 3 elementos.

```
1. public class App {
2.     public static void main(String[] args) {
3.
4.         Fila f = new Fila(3);
5.
6.         System.out.printf("Fila atual: %s.\n",
7. f.toString());
8.
9.         try {
10.             f.enfileira(29);
11.             System.out.printf("Fila %s <= %s
12. enfileirado.\n", f.toString(), 29);
13.             f.enfileira(10);
14.             System.out.printf("Fila %s <= %s
15. enfileirado.\n", f.toString(), 10);
16.         } catch (Exception e) {
17.             e.printStackTrace();
18.         }
19.     }
20. }
```

```

13.             f.enfileira(83);
14.             System.out.printf("Fila %s <= %s
enfileirado.\n", f.toString(), 83);
15.             f.enfileira(36);
16.             System.out.printf("Fila %s <= %s
enfileirado.\n", f.toString(), 36);
17.         } catch (Exception e) {
18.             System.err.println(e.getMessage());
19.         }
20.
21.         System.out.printf("Fila atual: %s.\n",
f.toString());
22.
23.         try {
24.             System.out.printf("Desenfileirando %d
<= fila %s.\n", f.desenfileira(), f.toString());
25.             System.out.printf("Desenfileirando %d
<= fila %s.\n", f.desenfileira(), f.toString());
26.             System.out.printf("Desenfileirando %d
<= fila %s.\n", f.desenfileira(), f.toString());
27.             System.out.printf("Desenfileirando %d
<= fila %s.\n", f.desenfileira(), f.toString());
28.         } catch (Exception e) {
29.             System.err.println(e.getMessage());
30.         }
31.     }
32. }

```

Na linha 4, é declarado o objeto da Fila com a capacidade máxima de 3 elementos. Entre as linhas 8 e 19, o primeiro bloco *try* é declarado. Utiliza-se esse bloco para tratar a exceção que pode ser lançada quando a fila estiver cheia. Dentro do bloco *try* existe a tentativa de inserir 4 elementos. Na quarta tentativa, uma exceção é lançada (linha 15) e tratada no bloco *catch* (linhas 17 a 19). A vantagem do tratamento de exceção é o que o código continua sua execução até o final.

A fila completa é exibida na linha 21. Depois, entre as linhas 23 e 30, é realizado o processo para desenfileirar os elementos. Novamente, 4 chamadas são realizadas e na quarta (linha 27), a exceção é lançada e tratada entre as linhas 28 e 30.

A saída para o código implementado na classe App é apresentado a seguir:

```
Fila atual: [].  
Fila [29] <= 29 enfileirado.  
Fila [29, 10] <= 10 enfileirado.  
Fila [29, 10, 83] <= 83 enfileirado.  
Fila atual: [29, 10, 83].  
A fila está com a capacidade máxima.  
Desenfileirando 29 <= fila [10, 83].  
Desenfileirando 10 <= fila [83].  
Desenfileirando 83 <= fila [].  
A fila está vazia.
```

1.1 Resolvendo o problema de capacidade máxima

A fila implementada com o vetor primitivo em Java funciona e atende a todos os requisitos dessa estrutura de dados. Contudo, ao se declarar de maneira primitiva, a fila apresenta o problema de capacidade máxima. Isso significa que caso exista, durante o tempo de execução do programa, a necessidade de se aumentar a capacidade dela, uma nova fila de capacidade maior deverá ser criada. Depois, todos os elementos da fila menor serão copiados para a maior. O desempenho para aumentar a capacidade de uma fila existente é bem questionável, sem contar que pode haver uma falha na cópia e a sequência ser perdida.

Para solucionar esse problema, pode-se optar pela coleção `ArrayList` existente na API `Collections` do Java. Ela se comporta como um vetor, porém a alocação de memória é totalmente dinâmica, não existindo mais a necessidade de ter uma capacidade máxima para a fila. Para atender a essa necessidade, a classe `Fila` será alterada. Sua nova implementação é apresentada a seguir:

```

1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class Fila {
5.     private final List<Integer> dados = new
ArrayList<Integer>();
6.
7.     public int primeiro() {
8.         return this.dados.get(0);
9.     }
10.
11.    public boolean ehVazia() {
12.        return this.dados.isEmpty();
13.    }
14.
15.    public int tamanho() {
16.        return this.dados.size();
17.    }
18.
19.    public void enqueue(int numero) {
20.        this.dados.add(numero);
21.    }
22.
23.    public int dequeue() {
24.        if(this.ehVazia()) {
25.            throw new RuntimeException("A fila
está vazia.");
26.        }
27.        return this.dados.remove(0);
28.    }
29.
30.    @Override
31.    public String toString() {
32.        return this.dados.toString();
33.    }
34. }

```

O primeiro passo da nova implementação é a inclusão da classe `ArrayList` e da interface `List` no cabeçalho da classe (linhas 1 e 2). Como `List` é uma interface, ela deve ser instanciada como um `ArrayList`, conforme implementado na linha 5.

O *ArrayList dados* é o único atributo necessário. Os métodos da classe *ArrayList* dão o suporte para os demais métodos da fila. O método *primeiro* (linhas 7 a 9) utiliza o método *get* de *ArrayList* para consultar o elemento na posição 0, que representa o primeiro elemento da fila. Entre as linhas 11 e 13, encontra-se o método *ehVazia*, que utiliza o próprio recurso do *ArrayList* verificando se a fila está vazia ou não. O método *tamanho* (linhas 15 a 17) também utiliza o recurso do método *size* implementado na classe *ArrayList* para retornar a quantidade de elementos armazenados na fila.

Mas, sem dúvidas, a simplificação maior está nos métodos *enfileira* (linhas 19 a 21) e *desenfileira* (linhas 23 a 28). Com o *ArrayList*, não há mais a necessidade de controlar a capacidade máxima da fila. Sendo assim, a verificação antes necessária de fila cheia é removida e o método *add* do *ArrayList* é utilizado para inserir o elemento ao final da fila, e, com isso, o método *desenfileira*, apesar de manter a verificação de fila vazia (linhas 24 a 26), só necessita do método *remove*, que recebe como parâmetro a posição da qual deve ser removido o elemento. Nesse caso, a posição é fixa e igual a 0, que representa o primeiro elemento da fila. O método *toString*, implementado entre as linhas 30 e 33, foi utilizado apenas para facilitar a exibição da fila, quando exibida no console.

1.2 Analisando o desempenho das operações

Apesar de solucionar o problema da capacidade, a implementação com *ArrayList* não soluciona o problema de desempenho do método *desenfileira*. Em ambas as implementações, esse método leva um tempo linear para mover todos os elementos para as primeiras posições após a remoção do primeiro elemento. Isso é uma desvantagem para esse tipo de implementação.

É importante lembrar que as operações realizadas por uma estrutura de dados não devem superar a complexidade de $O(\log n)$ e idealmente devem se manter em complexidade $O(1)$ (GOODRICH; TAMASSIA,

2013). O quadro 1 apresenta a complexidade de cada um dos métodos da classe Fila. Nesse caso, serve para ambas as implementações.

Quadro 1 – Apresentação das complexidades para as operações realizadas por cada um dos métodos

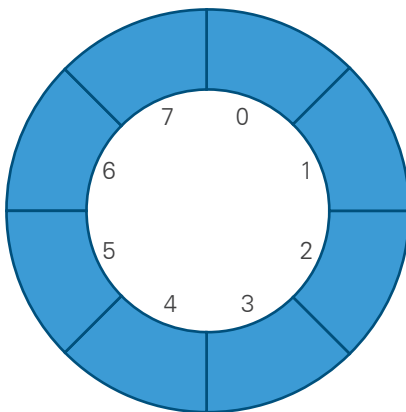
MÉTODO	COMPLEXIDADE
<i>primeiro</i>	$O(1)$
<i>ehVazia</i>	$O(1)$
<i>tamanho</i>	$O(1)$
<i>enfileira</i>	$O(1)$
<i>desenfileira</i>	$O(n)$

No quadro 1, é possível perceber que a maioria dos métodos utilizados executam as operações em complexidade constante. Apenas o método *desenfileira*, que tem um desempenho linear, quando comparado aos demais apresenta um desempenho ruim. A solução para melhorar o desempenho da classe Fila será apresentada nos tópicos a seguir.

2 Andando em círculo

Uma maneira para solucionar o desempenho linear do método *desenfileira* é trabalhar com uma lista circular. Dessa maneira, é possível controlar o início e o fim da fila de modo mais preciso. Contudo, essa implementação apresenta uma limitação, pois ela necessita de uma implementação com capacidade máxima. A figura 2 apresenta uma visão abstrata de uma lista circular de 8 posições, que serve como base para esse tipo de estrutura de dados.

Figura 2 – Apresentação abstrata de uma lista circular, que pode ser utilizada para armazenar as informações de uma fila



Com a lista circular, é necessário colocar mais um atributo para determinar as posições de início e fim e as demais informações de suporte. Dessa maneira, apesar de um ganho na velocidade das operações, é preciso estar ciente de que há uma perda em espaço de memória devido à quantidade de informações que serão armazenadas apenas para controle da estrutura de dados.

Veja a seguir a implementação da lista circular para atender à estrutura de dados tipo fila, utilizando a linguagem de programação Java.

```
1. public class Fila {  
2.     private final int[] dados;  
3.     private final int capacidade;  
4.     private int primeiro;  
5.     private int ultimo;  
6.     private int tamanho;  
7.  
8.     public Fila(int capacidade) {  
9.         this.capacidade = capacidade;  
10.        this.dados = new int[this.capacidade];  
11.    }
```

```

11.         this.primeiro = 0;
12.         this.ultimo = 0;
13.     this.tamanho = 0;
14.     }
15.
16.     public Fila() {
17.         this(10);
18.     }
19.
20.     public int primeiro() {
21.         return this.dados[this.primeiro];
22.     }
23.
24.     public boolean ehVazia() {
25.         return primeiro == ultimo && this.tamanho == 0;
26.     }
27.
28.     public int tamanho() {
29.         return this.tamanho;
30.     }
31.
32.     public void enqueue(int numero) {
33.         if(this.tamanho == this.capacidade) {
34.             throw new RuntimeException("A fila está
com a capacidade máxima.");
35.         }
36.         this.dados[this.ultimo] = numero;
37.         this.ultimo = (this.ultimo + 1) % this.
capacidade;
38.         this.tamanho++;
39.     }
40.
41.     public int dequeue() {
42.         if(this.ehVazia()) {
43.             throw new RuntimeException("A fila está
vazia.");
44.         }
45.         int numero = this.dados[this.primeiro];
46.         this.dados[this.primeiro] = -1;
47.         this.primeiro = (this.primeiro + 1) % this.
capacidade;
48.         this.tamanho--;
49.         return numero;

```

```
50.     }
51.
52.     @Override
53.     public String toString() {
54.         return Arrays.toString(this.dados);
55.     }
56. }
```

Entre as linhas 2 e 6, são apresentados os atributos necessários para pôr em prática a implementação da fila com uma lista circular. Os atributos apresentados são: o vetor primitivo de dados, já que existe a necessidade de manter a capacidade máxima de informações; o atributo da capacidade máxima; a posição do primeiro elemento da fila; e a posição do último elemento. Por fim, o atributo com o tamanho atual da fila.

Nesse momento, o foco será dado aos métodos *enfileira* (linhas 32 a 39) e *desenfileira* (linhas 41 a 50). Esses métodos são os que sofrem as maiores alterações na implementação em uma lista circular. Após a verificação se a fila está cheia, o elemento é atribuído na posição armazenada no atributo *ultimo*. Na sequência, o valor de *ultimo* é alterado para a expressão $(ultimo + 1) \text{ MOD } capacidade$ (linha 37). A operação de mod (em linguagem Java utiliza-se o operador %) retorna um número inteiro que é o resto da divisão da posição *ultimo* + 1 pela capacidade total do vetor. Essa técnica utilizada com o mod auxilia manter os valores da posição dentro do índices do vetor (GOODRICH; TAMASSIA, 2013). Na sequência, o tamanho é incrementado em 1.

O método *desenfileira*, após a validação da fila vazia, utiliza uma variável auxiliar *numero* (linha 45) para armazenar o número que será removido. Depois, a posição recebe o valor de -1, que representa um valor nulo para os números inteiros nessa aplicação. Isso garante que o local esteja disponível para novos armazenamentos. Na linha 47, a mesma operação de mod realizada na linha 37 é feita. Contudo, em vez de utilizar o atributo *ultimo*, essa operação é realizada com o atributo *primeiro*. Por fim, o tamanho deve ser decrementado e o número retornado.

Apesar da necessidade de mais informações na classe Fila, essa implementação é mais eficiente quando se considera o fator de tempo de processamento das informações. Com ela, todos os métodos nesse momento estão operando em complexidade $O(1)$.

3 Ligando os elementos

A maneira mais eficiente de implementar uma fila é utilizando uma lista ligada. Com esse tipo de lista, é possível chegar a um custo-benefício ideal entre a memória de armazenamento e o tempo de processamento ótimo. Entretanto, uma lista ligada depende de uma classe Elemento que contenha dois atributos, no caso, um *inteiro* para armazenar o valor (pode ser outro tipo, o inteiro é só para se manter similar aos demais exemplos) e um atributo do tipo *elemento*, que deve armazenar o próximo elemento da lista. Como a fila possui operações limitadas, o uso da lista ligada nessa situação é a melhor opção. Veja a classe Elemento implementada a seguir.

```
1. public class Elemento {
2.     private int valor;
3.     private Elemento proximo;
4.
5.     public Elemento() {
6.         this.valor = -1;
7.         this.proximo = null;
8.     }
9.
10.    public Elemento(int valor) {
11.        this.valor = valor;
12.        this.proximo = null;
13.    }
14.
15.    public int getValor() {
16.        return valor;
17.    }
18.
19.    public void setProximo(Elemento proximo) {
20.        this.proximo = proximo;
```



```

21.     }
22.
23.     public Elemento getProximo() {
24.         return proximo;
25.     }
26. }

```

A classe `Elemento` é simples e possui apenas dois atributos, como explicado anteriormente, os métodos *getters* e *setters* e dois construtores, um simples e outro com o valor desejado que será inserido na fila de dados. Agora, é apresentada a classe `Fila` utilizando a classe `Elemento` como base da implementação.

```

1. public class Fila {
2.     private Elemento primeiro = null;
3.     private Elemento ultimo = null;
4.     private int tamanho = 0;
5.
6.     public Elemento primeiro() {
7.         return this.primeiro;
8.     }
9.
10.    public boolean ehVazia() {
11.        return this.tamanho == 0;
12.    }
13.
14.    public int tamanho() {
15.        return this.tamanho;
16.    }
17.
18.    public void enqueue(int numero) {
19.        if (this.primeiro == null) {
20.            this.primeiro = new Elemento(numero,
21.            null);
22.        } else if (this.ultimo == null) {
23.            this.ultimo = new Elemento(numero,
24.            null);
25.            this.primeiro.setProximo(this.ultimo);

```

```

24.         } else {
25.             Elemento e = new Elemento(numero,
null);
26.             this.ultimo.setProximo(e);
27.             this.ultimo = new Elemento();
28.             this.ultimo = e;
29.         }
30.         this.tamanho++;
31.     }
32.
33.     public Elemento desenfileira() {
34.         if (this.ehVazia()) {
35.             throw new RuntimeException("A fila
está vazia.");
36.         }
37.         Elemento e = this.primeiro;
38.         this.primeiro = this.primeiro.getProximo();
39.         this.tamanho--;
40.         if (this.tamanho == 0) {
41.             this.primeiro = null;
42.             this.ultimo = null;
43.         }
44.         return e;
45.     }
46. }

```

Apenas 3 atributos são necessários, o primeiro elemento (linha 2), o último elemento (linha 3), ambos inicializados com valores nulos, e também o controle do tamanho em que se encontra a fila (linha 4). Entre as linhas 18 e 31, é apresentado o método *enfileira*. Existem três possibilidades para enfileirar um elemento, a primeira é quando a fila está vazia, validada pelo primeiro elemento igual a nulo (linha 19). A segunda possibilidade é quando existe apenas um elemento na fila, validada pelo atributo *ultimo* igual a nulo (linha 21). Por fim, a partir de dois elementos na fila, o atributo *proximo* da referência armazenada em *ultimo* recebe o novo elemento (linha 26). Uma nova referência é criada para o atributo *ultimo* (linha 27), assim é possível deixá-lo independente na fila,

evitando a perda de informações. Por fim, o novo elemento é copiado para o atributo *ultimo* (linha 28). Em tempo constante, a operação de enfileirar é executada pelo algoritmo.

O método *desenfileira* é apresentado entre as linhas 33 e 45. A primeira parte do método, como padrão, é verificar se a fila está vazia. Na sequência, separa-se o elemento que será removido em um objeto novo (linha 37). O atributo *primeiro* recebe o próximo elemento para ocupar o primeiro lugar na fila (linha 38). O tamanho deve ser decrementado em 1 e, caso chegue a 0, tanto o atributo *primeiro* como o atributo *ultimo* devem receber o valor de nulo indicando uma fila vazia novamente (entre as linhas 39 e 43). Por fim, o objeto auxiliar é retornado pelo método. Perceba que esse método também tem sua complexidade constante. Comparando a classe com a implementação através de uma lista ligada, pode-se dizer que é o melhor custo benefício entre as demais implementações. Isso acontece para se manter o equilíbrio entre a quantidade de informações necessárias para torná-la operacional e também para que todas suas operações se mantenham em $O(1)$, tempo constante. O quadro 2 apresenta a complexidade de cada método da classe Fila utilizando como base uma lista ligada.

Quadro 2 – Apresentação das complexidades para as operações realizadas por cada um dos métodos, agora utilizando como base a lista ligada

MÉTODO	COMPLEXIDADE
<i>primeiro</i>	$O(1)$
<i>ehVazia</i>	$O(1)$
<i>tamanho</i>	$O(1)$
<i>enfileira</i>	$O(1)$
<i>desenfileira</i>	$O(1)$

O Java na API Collection já possui uma interface de fila chamada Queue, evitando assim a necessidade de criar a estrutura toda do zero. Mas é importante saber como funcionam todas as partes das implementações de uma estrutura de dados qualquer, pois, muitas vezes, é necessário adaptá-la para atender a algum problema específico.



PARA SABER MAIS

Para se aprofundar mais sobre o uso da interface Queue disponibilizada pelo próprio Java, acesse o link: <https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html> (acesso em: 20 abr. 2020).

4 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Utilizando o método `System.currentTimeMillis()`, disponível no Java, compare o tempo de processamento dos métodos *desenfileira* apresentados ao longo do capítulo.
2. Crie duas filas de atendimento bancário, uma com atendimento normal e outra de atendimento prioritário. A fila de atendimento prioritário deve ser duas vezes mais rápida que a de atendimento normal. Teste sua implementação.
3. Pesquise a aplicação de fila em escalonadores *round-robin*.
4. Implemente um escalonador *round-robin* para um sistema de logística em que é necessário carregar caminhões para entrega.

Considerações finais

Apresentamos ao longo deste capítulo o conceito de fila e como ele é importante para a aplicação em diversos sistemas, inclusive em tarefas computacionais triviais. Algumas das possíveis implementações foram apresentadas e discutidas, demonstrando as vantagens e desvantagens de cada uma delas. Sabe-se também que a própria API Collections do Java já possui uma implementação de fila de maneira nativa, assim como diversas outras linguagens de programação. Alinhando o conhecimento da estrutura de dados com os algoritmos, muitas possibilidades devem ser exploradas para otimizações e soluções de novos problemas.

Referências

CORMEN, Thomas H. *et al.* **Introduction to algorithms**. Cambridge: MIT Press, 2009.

DEITEL, Paul J.; DEITEL, Harvey M. **Java: como programar**. São Paulo: Pearson, 2008.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

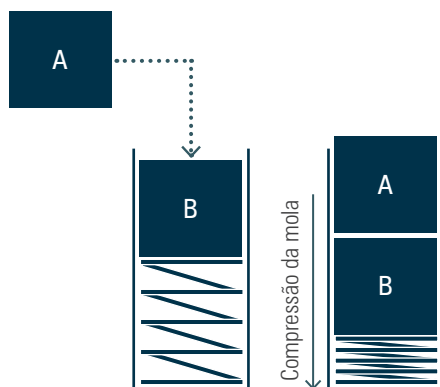
NECAISE, Rance D. **Data structures and algorithms using Python**. Hoboken: John Wiley & Sons, Inc., 2010.

Pilha

Quando nos mudamos para uma nova residência, normalmente colocamos os nossos pertences em caixas para facilitar seu transporte. Enquanto os arrumamos, podemos colocar uma caixa em cima de outra, até um certo limite de peso. O ato de colocar uma caixa sobre a outra é conhecido como empilhar. Quando vamos mudar essas caixas de lugar, deve-se sempre retirar primeiro a caixa que está por cima de todas as demais e assim por diante. As caixas posicionadas da maneira descrita são chamadas de pilha. Uma pilha pode existir em diversos cenários do cotidiano.

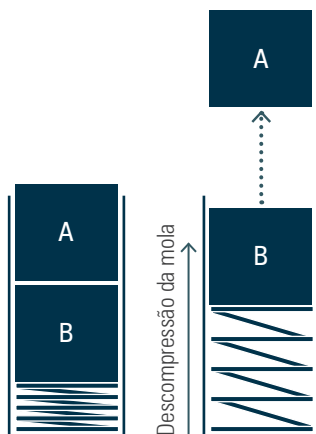
Alguns exemplos de aplicações de pilha no cotidiano: uma pilha de pratos para lavar; a organização das entregas dentro dos caminhões das empresas de logística; os sistemas de solução de equações matemáticas; os dispensadores de objetos, como nas máquinas de venda automática; entre muitos outros cenários (NECAISE, 2010; GOODRICH; TAMASSIA, 2013). Para exemplificar o modelo de uma pilha, vamos considerar o funcionamento de uma máquina de venda automática. Ela possui diversos compartimentos nos quais os produtos são armazenados. Ao fundo do compartimento, há uma mola que é comprimida de maneira controlada e, quando é realizada uma compra, a mola empurra para fora o produto que está na ponta, entregando-o a quem o comprou. Esse exemplo é demonstrado na figura 1.

Figura 1 – Processo de abastecimento do compartimento de uma máquina de venda automática (esse processo também é conhecido como empilhar objetos)



A remoção do produto desse compartimento é realizada de maneira contrária ao processo de abastecimento ou armazenamento do produto. Para facilitar a visualização do processo, ele é demonstrado na figura 2.

Figura 2 – Processo de retirada do produto A do compartimento da máquina de venda automática (esse processo também é conhecido como desempilhar um objeto)



Na computação, quando se utiliza uma pilha como estrutura de dados de um sistema qualquer, ela é considerada como uma estrutura do tipo LIFO (*last-in first-out*, em inglês). Em outras palavras, a pilha é uma estrutura de dados em que o último objeto inserido no vetor será o primeiro a deixá-lo. A pilha, em sua essência, não deixa de ser uma lista de objetos, porém suas operações são limitadas de acordo com as regras da categoria LIFO (FORBELLONE; EBERSPACHER, 2005; CORMEN *et al.*, 2009; NECAISE, 2010; GOODRICH; TAMASSIA, 2013).

Na definição de uma estrutura do tipo pilha, devem ser considerados os métodos básicos para se determinar as operações que podem ser realizadas por ela (DEITEL; DEITEL, 2008; NECAISE, 2010). Os métodos que devem ser considerados são:

- **ehVazia (*isEmpty*, em inglês):** informa um resultado booleano dizendo se a pilha possui algum elemento armazenado ou não.
- **tamanho (*size*, em inglês):** retorna a quantidade de elementos armazenados, no momento da consulta, dentro da pilha.

- **topo (top, em inglês):** consulta qual o elemento está na posição do topo da pilha. Esse método não deve remover o elemento, ele é apenas informativo.
- **empilha (push, em inglês):** insere um elemento no topo da pilha. Esse método deve receber como parâmetro o elemento que é inserido na pilha.
- **desempilha (pop, em inglês):** remove e retorna o elemento que está no topo da pilha. Uma regra importante nesse método é que nenhum elemento pode ser removido de uma pilha vazia, portanto uma mensagem de erro deve retornar caso isso ocorra (GOODRICH; TAMASSIA, 2013).

Esses são os métodos básicos para a implementação de uma pilha. Como base para o armazenamento dos dados é utilizada uma lista comum de controle com operações específicas. A implementação da pilha pode ser realizada de três maneiras pelo menos: com um vetor estático, um vetor dinâmico ou uma lista ligada. Cada implementação será discutida e apresentada nos próximos tópicos.

1 Implementando a pilha com o uso de vetores

Quando se utiliza Java como linguagem de programação para se implementar uma pilha, é possível trabalhar com dois tipos de vetores. A primeira possibilidade é o uso de um vetor primitivo, que possui algumas limitações de implementação. A outra possibilidade é o uso de um vetor oferecido pela API Collections do Java. Com a segunda possibilidade, existem alguns métodos de suporte que facilitam o processo de implementação da pilha, além de possuir um armazenamento de memória dinâmico, facilitando o controle da quantidade de elementos

inseridos na pilha. Em outras palavras, o limite é a quantidade de memória disponível no hardware no qual o sistema será executado.

Para melhor entendimento sobre os métodos básicos de uma pilha, a primeira implementação é realizada utilizando um vetor primitivo de inteiros. A classe Pilha implementada deve receber em seu construtor a capacidade máxima de objetos armazenados. Os cinco métodos básicos são implementados para que a classe fique completa. Atributos de suporte e controle da pilha também são declarados na implementação da classe. Outro ponto importante é que uma vez utilizados vetores primitivos na implementação, é preciso verificar se a capacidade máxima da pilha foi atingida antes de inserir qualquer objeto no topo. Essas validações permitem que a pilha possa ser aplicada em quaisquer sistemas, sem que ocorra algum problema na execução. A classe Pilha é apresentada a seguir, sendo sua implementação com um vetor primitivo de inteiros para simplificar a explicação.

```
1. import java.util.Arrays;
2.
3. public class Pilha {
4.     private int[] dados;
5.     private int topo;
6.     private int capacidade;
7.
8.     public Pilha(int capacidade) {
9.         this.capacidade = capacidade;
10.        this.dados = new int[this.capacidade];
11.        this.topo = -1;
12.    }
13.
14.    public Pilha() {
15.        this(10);
16.    }
17.
18.    public boolean ehVazia() {
19.        return topo == -1;
```

```

20.     }
21.
22.     public int topo() {
23.         return this.dados[this.topo];
24.     }
25.
26.     public int tamanho() {
27.         return this.topo + 1;
28.     }
29.
30.     public void empilha(int numero) {
31.         if ((this.topo + 1) >= this.capacidade) {
32.             throw new RuntimeException("A pilha
está cheia.");
33.         }
34.         this.dados[++this.topo] = numero;
35.     }
36.
37.     public int desempilha() {
38.         if(this.ehVazia()) {
39.             throw new RuntimeException("A pilha
está vazia.");
40.         }
41.         return this.dados[this.topo--];
42.     }
43.
44.     @Override
45.     public String toString() {
46.         return Arrays.toString(this.dados);
47.     }
48. }

```

A classe *Pilha* apresentada possui 3 atributos: o vetor de inteiros, que armazena os elementos (*dados*); o atributo *topo*, que armazena a posição do elemento que está no topo da pilha; e um inteiro que mantém a capacidade total da pilha. Dois construtores foram construídos, o primeiro (linhas 8 a 12) recebe como parâmetro a capacidade total da pilha e, com essa informação, inicializa o vetor *dados*. Por fim, ele inicializa a posição *topo* com o valor de -1 , o que significa que a pilha está vazia. O

segundo construtor (linhas 14 a 16) apenas chama o primeiro construtor, informando uma capacidade padrão de 10 elementos, caso não seja informada uma capacidade. O método *ehVazia* (linhas 18 a 20) retorna o resultado da condição lógica na comparação de igualdade entre o *topo* e o valor -1 . O método *topo* (linhas 22 a 24) retorna o valor armazenado na posição *topo*, apenas como consulta. O método *tamanho* (linhas 26 a 28) retorna o valor da posição *topo* + 1, pois, somando 1 a qualquer índice de um vetor, obtém-se a ordem do elemento no vetor.

O método *empilha*, entre as linhas 30 e 35, recebe como parâmetro um número que será armazenado na pilha. Entre as linhas 31 e 33, é realizado um teste para validar se a pilha não está cheia. Caso seja verdade, uma exceção do tipo *RuntimeException* é lançada com a mensagem de pilha cheia. Se a pilha não está cheia, é incrementado 1 na posição *topo*, e na sequência o valor de número é armazenado no topo da pilha. A instrução *++this.topo* realiza primeiro o incremento para depois utilizar o valor armazenado em *topo*. É um comportamento diferente da instrução *this.topo++*, que primeiro utiliza o valor armazenado na variável para depois incrementá-la. O método *desempilha* é implementado entre as linhas 37 e 42. Há a validação se existe algum elemento na pilha, antes de realizar o processo de retirada. Em caso de pilha vazia, uma exceção do tipo *RuntimeException* é lançada com a mensagem "A pilha está vazia.". Em seguida, na linha 41, é retornado o valor do *topo* subtraído em 1. Para concluir a implementação da classe, entre as linhas 44 e 47, o método *toString* é sobrescrito para facilitar a exibição dos elementos armazenados na pilha.

A classe App apresentada a seguir faz o teste da classe Pilha, com uma capacidade máxima igual a 3 elementos.

```
1. public class App {  
2.     public static void main(String[] args) {  
3.
```

```

4.         Pilha p = new Pilha(3);
5.
6.         System.out.printf("Pilha vazia: %s\n" ,
p.toString());
7.
8.         try {
9.             p.empilha(29);
10.            System.out.printf("Empilha: %s <=
%d\n", p.toString(), 29);
11.            p.empilha(42);
12.            System.out.printf("Empilha: %s <=
%d\n", p.toString(), 42);
13.            p.empilha(10);
14.            System.out.printf("Empilha: %s <=
%d\n", p.toString(), 10);
15.            p.empilha(83);
16.            System.out.printf("Empilha: %s <=
%d\n", p.toString(), 83);
17.        } catch (RuntimeException e) {
18.            System.err.println(e.getMessage());
19.        }
20.
21.        System.out.printf("Pilha cheia: %s\n" ,
p.toString());
22.
23.        try {
24.            System.out.printf("Desempilha: %s =>
%d\n", p.toString(), p.desempilha());
25.            System.out.printf("Desempilha: %s =>
%d\n", p.toString(), p.desempilha());
26.            System.out.printf("Desempilha: %s =>
%d\n", p.toString(), p.desempilha());
27.            System.out.printf("Desempilha: %s =>
%d\n", p.toString(), p.desempilha());
28.        } catch (RuntimeException e) {
29.            System.err.println(e.getMessage());
30.        }
31.    }
32. }

```

Na classe `App`, uma instância da classe `Pilha` é criada com a capacidade máxima igual a 3 elementos (linha 4). Na sequência, é apresentada a pilha vazia, a título de confirmação (linha 6). Da linha 8 até a 19, é declarado o primeiro bloco `try`, no qual o processo de empilhar os elementos é realizado. Esse processo é realizado até que a pilha esteja cheia, momento em que é feita mais uma tentativa para gerar uma exceção com a mensagem de “A pilha está cheia” (linha 15). A exceção é tratada apenas exibindo a mensagem retornada do lançamento da exceção (linha 18). Para fins de confirmação, a pilha cheia é exibida, de acordo com o código apresentado na linha 21. O segundo bloco `try` é apresentado entre as linhas 23 e 30. São realizadas 4 chamadas para o método `desempilhar`. A quarta chamada (linha 27) é gerada, e, como a pilha está vazia, uma exceção é lançada. Na linha 29, a exceção é tratada e a mensagem “A pilha está vazia.” é exibida ao usuário.

A saída para o código implementado na classe `App` é apresentada a seguir:

```
Pilha vazia: []
Empilha:    [29] <= 29
Empilha:    [29, 42] <= 42
Empilha:    [29, 42, 10] <= 10
A pilha está cheia.

Pilha cheia: [29, 42, 10]
Desempilha:  [29, 42] => 10
Desempilha:  [29] => 42
Desempilha:  [] => 29
A pilha está vazia.
```

1.1 Ampliando a capacidade da pilha

A implementação anterior da pilha delimita a sua capacidade no momento da instância do objeto, graças ao vetor primitivo utilizado pelo

Java. Apesar disso, todos os requisitos descritos para uma estrutura de dados em formato de pilha são atendidos. Porém, quando se utiliza um vetor primitivo, expandir a capacidade máxima em qualquer projeto pode ser uma tarefa custosa e até a mão de obra de um desenvolvedor pode ser necessária. Uma solução possível é copiar os elementos da pilha para uma pilha temporária, criar uma nova instância do tamanho desejado, e reinserir os elementos na pilha com a capacidade aumentada. A nova pilha deve ser construída com o máximo de cautela, pois dados podem se perder ao longo desse processo.

A solução mais aconselhada para deixar a capacidade da pilha dinâmica é o uso de uma lista disponível através da API Collections do próprio Java. O objeto mais aconselhável para essa finalidade é o `ArrayList`. Seu comportamento nativo é como o do vetor primitivo, contudo, a alocação de memória realizada por ele é dinâmica. Isso significa que durante a execução do programa, caso seja necessário, o próprio Java aloca um novo espaço na memória para continuar a execução. Assim, a implementação da classe `Pilha` é facilitada, uma vez que não há a necessidade de se manter um atributo para a capacidade da pilha, nem mesmo de se verificar se a pilha está cheia no método *empilha*. A classe `Pilha` é alterada para incluir o `ArrayList` em seu código. Veja o resultado da implementação no código a seguir.

```
1. import java.util.ArrayList;
2. import java.util.List;
3.
4. public class Pilha {
5.     private List<Integer> dados = new
ArrayList<Integer>();
6.
7.     public boolean ehVazia() {
8.         return this.dados.isEmpty();
9.     }
10.
```



```
11.     public int topo() {
12.         return this.dados.get(this.dados.size() -
13.     );
14.     }
15.     public int tamanho() {
16.         return this.dados.size();
17.     }
18.
19.     public void empilha(int numero) {
20.         this.dados.add(numero);
21.     }
22.
23.     public int desempilha() {
24.         if(this.ehVazia()) {
25.             throw new RuntimeException("A pilha
está vazia.");
26.         }
27.         return this.dados.remove(this.dados.size() -
28.     );
29.     }
30.     @Override
31.     public String toString() {
32.         return this.dados.toString();
33.     }
34. }
```

A classe `ArrayList` e a interface `List` são importadas para uso como meio de armazenamento dos dados (linhas 1 a 2). O uso do `ArrayList` torna necessário apenas um atributo, que é a instância de seu objeto, o `ArrayList dados` (linha 5). Os demais atributos utilizados na implementação com o vetor primitivo não são mais necessários, uma vez que a classe `ArrayList` provê essas informações.

O método `ehVazia` (linhas 7 a 9) retorna o valor booleano gerado pela função `isEmpty` do `ArrayList`. Entre as linhas 11 e 13, é apresentado o método `topo`. Ele retorna o valor armazenado na última posição do

atributo *dados*. Para essa implementação, utiliza-se o método *get*, que recebe o índice da posição que se deseja consultar. O cálculo do índice é feito com a função *size* do atributo *dados*, que informa a quantidade de elementos armazenados no *ArrayList*. Encontrar o índice a partir do valor de *size* é simples, basta subtrair 1, assim os índices são considerados a partir do valor 0 e encontra-se o último elemento. Ele representa o topo da pilha. O método *tamanho* (linhas 15 a 17) retorna o valor gerado pelo método *size* de dados. Com ele, é determinado o valor de quantos elementos estão na pilha.

No método *empilha* (linhas 19 a 21) não se faz necessário a validação da capacidade máxima da pilha. Portanto, basta utilizar o método *add* para inserir um elemento na última posição de dados. A última posição representa exatamente o topo da pilha. Em seguida, implementa-se o método *desempilha* (linhas 23 a 28). É necessário validar se existe algum elemento para ser desempilhado, para isso, utiliza-se o método *ehVazia* (linhas 24 a 26). Na linha 27, é feito o retorno do valor que está no topo da pilha, a última posição de dados. O truque para encontrar a última posição é o mesmo utilizado no método *topo*, só que, em vez de utilizar o método *get* de dados, é o método *remove* que é invocado. Por fim, o método *toString* é implementado com a invocação do método *toString* do *ArrayList*. Assim, a visualização da pilha na linha de comando ou console é facilitada.

2 Utilizando uma lista ligada para empilhar objetos

Em questão de desempenho, o da pilha implementada com o *ArrayList* é bem similar ao de uma lista ligada. Contudo, o uso da lista ligada apresenta a vantagem de se manter o controle em todos os passos e métodos de um processo de estrutura de dados em formato de pilha. A implementação com uma lista ligada precisa de uma outra classe de apoio. Essa classe representa o elemento que será armazenado pela

pilha. Nela são declarados dois atributos: o valor do tipo da pilha e um atributo recursivo da própria classe que simboliza o próximo elemento da pilha. A classe Elemento é apresentada a seguir:

```
1.  public class Elemento {
2.      private int valor;
3.      private Elemento proximo;
4.
5.      public Elemento() {
6.          this.valor = -1;
7.          this.proximo = null;
8.      }
9.
10.     public Elemento(int valor, Elemento proximo) {
11.         this.valor = valor;
12.         this.proximo = proximo;
13.     }
14.
15.     public int getValor() {
16.         return valor;
17.     }
18.
19.     public void setProximo(Elemento proximo) {
20.         this.proximo = proximo;
21.     }
22.
23.     public Elemento getProximo() {
24.         return proximo;
25.     }
26. }
```

A classe Elemento é simples e possui apenas dois atributos, como explicado anteriormente, os métodos *getters* e *setters* e dois construtores, um simples e outro com o valor desejado a ser inserido na pilha. O tipo do valor foi escolhido como um número inteiro para facilitar a explicação, mas pode ser aplicado com qualquer outra classe. Na sequência, a implementação da classe Pilha, no formato de uma lista ligada, é apresentada.

```

1.  public class Pilha {
2.      private Elemento topo;
3.      private int tamanho;
4.
5.      public boolean ehVazia() {
6.          return this.topo == null;
7.      }
8.
9.      public Elemento topo() {
10.         return this.topo;
11.     }
12.
13.     public int tamanho() {
14.         return this.tamanho;
15.     }
16.
17.     public void empilha(int numero) {
18.         Elemento novo = new Elemento(numero, null);
19.         if(this.ehVazia()) {
20.             this.topo = novo;
21.         } else {
22.             novo.setProximo(this.topo);
23.             this.topo = new Elemento();
24.             this.topo = novo;
25.         }
26.         this.tamanho++;
27.     }
28.
29.     public Elemento desempilha() {
30.         if(this.ehVazia()) {
31.             throw new RuntimeException("A pilha
está vazia.");
32.         }
33.         Elemento removido = this.topo;
34.         this.topo = this.topo.getProximo();
35.         this.tamanho--;
36.         return removido;
37.     }
38.
39.     @Override
40.     public String toString() {

```

```

41.         StringBuilder sb = new StringBuilder("[");
42.         if(!this.ehVazia()) {
43.             Elemento e = this.topo;
44.             while (e != null) {
45.                 sb.append(e.getValor());
46.                 if (e.getProximo() != null)
sb.append(", ");
47.                 e = e.getProximo();
48.             }
49.         }
50.         sb.append("]");
51.         return sb.toString();
52.     }
53. }

```

Dois atributos são declarados: um elemento que representa o *topo* (linha 2); e um inteiro que representa o *tamanho* da pilha (linha 3). Os métodos *ehVazia*, *tamanho* e *topo* fazem uma função de *getters* para os atributos da classe Pilha. As implementações mais complexas estão nos três métodos seguintes: *empilha*, *desempilha* e *toString*, sendo o último opcional para o funcionamento da estrutura de dados. Entre as linhas 17 e 27, encontra-se o método *empilha*. A partir do número recebido como parâmetro pelo método *empilha*, um novo elemento é gerado (linha 18). Na sequência, é verificado se a pilha está vazia, e, em caso positivo, o elemento gerado é atribuído ao topo (linha 20). Caso contrário, o topo é atribuído com o valor do próximo elemento gerado, e o novo elemento passa a ser o novo topo da pilha (linhas 22 a 24). Por fim, o tamanho da pilha é incrementado em 1 (linha 26).

O próximo método encontrado na classe Pilha é o *desempilha* (linhas 29 a 37), que retorna um objeto do tipo Elemento. O primeiro passo do método é verificar se a pilha está vazia e lançar uma exceção em caso positivo (linhas 30 a 32). Existindo um elemento para sair da pilha, o método atribui a uma variável auxiliar o objeto que está no topo (linha 33). Depois ele faz com que o topo vire seu próximo elemento (linha

34). Subtrai 1 do tamanho da pilha (linha 35) e retorna o objeto que foi armazenado na variável auxiliar (linha 36). O último método da classe é o método *toString* (linhas 39 a 52). Ele é um método opcional, apenas para uma melhor visualização dos elementos da pilha. Como a estrutura da lista ligada é criada e não nativa do Java, é necessário criar a implementação toda do método. Nas implementações anteriores, por utilizar vetores, o Java já possui métodos auxiliares para esse suporte.

Apesar das implementações realizadas ao longo do capítulo, o Java possui na API Collections uma interface para pilha chamada Stack. Dessa maneira, a criação da estrutura totalmente do zero não é necessária. Mas o estudo de seu funcionamento é muito importante, pois existem cenários no dia a dia de qualquer sistema em que são necessárias personalizações nos códigos. Agora, com o conhecimento sobre o funcionamento e de como implementar uma pilha, as eventuais adaptações ocorrerão de forma natural.



PARA SABER MAIS

Para se aprofundar mais sobre o uso da interface Stack disponibilizada pelo próprio Java, acesse o link: <https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html> (acesso em: 20 abr. 2020).

3 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Utilizando o método `System.currentTimeMillis()` disponível no Java, compare o tempo de processamento dos métodos *empilha* apresentados ao longo do capítulo. Use também a interface Stack.

2. Utilizando o método `System.currentTimeMillis()` disponível no Java, compare o tempo de processamento dos métodos *desempilha* apresentados ao longo do capítulo. Use também a interface `Stack`.
3. Crie um sistema de logística que deverá gerar uma lista de entregas para um caminhão, no qual o primeiro pacote alocado no caminhão deverá ser o último a chegar ao destino. Considere também que o último destino é o local mais próximo da transportadora.
4. Crie uma simulação do sistema de desfazer uma alteração no editor de texto, ou seja, o comando gerado pela ação de pressionar as teclas `Control` e `Z`.

Considerações finais

Apresentamos ao longo deste capítulo o conceito de pilha e como ele é importante para aplicações em diversos sistemas, inclusive em tarefas triviais. Algumas das possíveis implementações foram apresentadas e discutidas demonstrando as vantagens e desvantagens de cada uma delas. Sabe-se também que a própria API `Collections` do Java possui uma implementação da pilha através da interface `Stack`, assim como diversas outras linguagens de programação, como Python, C#, JavaScript e muitas outras. Alinhando o conhecimento da estrutura de dados com os algoritmos, muitas possibilidades devem ser exploradas para otimizações e soluções de novos problemas.

Referências

CORMEN, Thomas H. *et al.* **Introduction to algorithms**. Cambridge: MIT Press, 2009.

DEITEL, Paul J.; DEITEL, Harvey M. **Java**: como programar. São Paulo: Pearson, 2008.

FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Prentice Hall, 2005.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

NECAISE, Rance D. **Data structures and algorithms using Python**. Hoboken: John Wiley & Sons, Inc., 2010.

Fila de prioridade e heap

Um sistema de mensageria é implementado. Esse sistema gerencia a comunicação entre diversos sistemas, criando uma certa integração entre eles, por exemplo, autoatendimento e mainframe, em instituições financeiras. Contudo, com o aumento do volume de mensagens, informações importantes são perdidas ao longo do processo. Esse é o típico caso para o uso de uma fila de prioridade.

Em linhas gerais, uma fila de prioridade tem o comportamento parecido com o de uma fila tradicional, a FIFO (*first-in first-out*, em inglês). Contudo, a fila tradicional, assim como os demais tipos de estrutura de dados, tem uma sequência arbitrária de entrada e saída, geralmente organizadas de maneira linear entre seus elementos armazenados (NECAISE, 2010; GOODRICH; TAMASSIA, 2013).

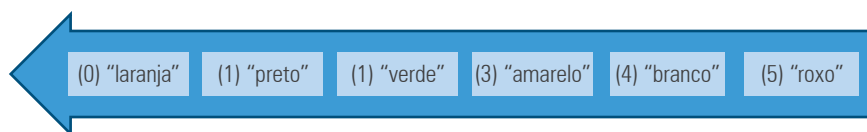
Diferentemente das estruturas tradicionais (apesar da entrada de uma fila de prioridade ser arbitrária), sua saída é determinada pela prioridade do elemento. Quanto maior a prioridade dele, mais rápido ele deve ser removido da fila (CORMEN *et al.*, 2009; GOODRICH; TAMASSIA, 2013).

Sendo assim, pode-se entender que a fila de prioridade é uma extensão da fila tradicional, na qual é atribuído um valor de prioridade para o elemento armazenado. Um elemento, ou classe, deve ser criado para que seja atribuída a prioridade de cada item da fila. Ele deve armazenar o objeto do tipo da fila e um valor inteiro que represente a sua prioridade. A prioridade mais alta de uma fila é dada pelo seu menor inteiro, em outras palavras, o valor 0 é o elemento de maior prioridade dentro da fila. Ele deve ser removido primeiro. No caso da existência de diversos itens com a mesma prioridade, é importante ressaltar que os itens devem se comportar como uma fila tradicional. Isso vale apenas para os itens com o mesmo grau de prioridade (NECAISE, 2010; GOODRICH; TAMASSIA, 2013). Veja uma sequência de pseudocódigo a seguir:

```
1.   enfileirar("roxo", 5)
2.   enfileirar("preto", 1)
3.   enfileirar("laranja", 0)
4.   enfileirar("branco", 4)
5.   enfileirar("verde", 1)
6.   enfileirar("amarelo", 3)
```

Cada linha do código acima representa uma chamada para enfileirar uma cor dentro da fila de prioridade. Como segundo parâmetro do método *enfileirar*, é informado o valor da prioridade da cor dentro da fila. Com base na sequência apresentada, a figura 1 apresenta a posição dos itens na fila de prioridade de acordo com as regras dessa estrutura de dados.

Figura 1 – Fila de prioridade entre cores apresentando a sequência de saída de acordo com a prioridade de cada item



Perceba que indiferentemente do momento em que cada item foi adicionado na fila, ela manteve a ordem de prioridade começando do inteiro 0 e, no exemplo, indo até o 5. Veja que cada item está representado pela prioridade entre parênteses e o valor posicionado logo na sequência da linha.

Quando ocorreu a situação de existirem dois itens com a mesma prioridade na figura 1, respeitou-se a ordem de inserção na fila. Esse é o caso que ocorreu entre os itens “preto” e “verde” na fila de prioridade apresentada.

Entre as filas de prioridade podem existir dois tipos, as restritas (*bounded*, em inglês) e as sem restrições (*unbounded*). Uma fila de prioridade restrita determina o intervalo de inteiros que representará os valores de prioridade. Nesse caso, não pode ser determinado nenhuma prioridade fora desse intervalo. Já a fila de prioridade sem restrições não tem um intervalo de prioridades predeterminado. Sendo assim, a única regra referente às prioridades é que o menor inteiro tem prioridade mais alta que o maior inteiro (NECAISE, 2010).

Na definição de uma estrutura de dados do tipo fila de prioridade, devem ser considerados os métodos básicos para determinar as operações que podem ser realizadas por ela (FORBELLONE; EBERSPACHER, 2014; NECAISE, 2010; GOODRICH; TAMASSIA, 2013). Os métodos que devem ser considerados são:

- **FilaPrioridade():** cria uma fila de prioridade sem restrições.

- **FilaPrioridade(limite):** cria uma fila de prioridade restrita, em que o intervalo é iniciado em 0 e vai até o limite informado como parâmetro.
- **ehVazia (isEmpty, em inglês):** informa um resultado booleano dizendo se a pilha possui algum elemento armazenado ou não.
- **tamanho (length, em inglês):** retorna à quantidade de elementos/ itens armazenados no momento da consulta.
- **enfileira (item, prioridade) (enqueue, em inglês):** insere um elemento/item na fila de acordo com a prioridade informada. A prioridade deve estar dentro do intervalo limitante, nos casos de filas restritas.
- **desenfileira (dequeue, em inglês):** remove e retorna o elemento que está no início da fila de prioridade. Esse elemento possui o maior grau de prioridade. É importante também verificar se a fila não está vazia antes de realizar essa operação.

Com os métodos básicos definidos e apresentados, é hora de implementar a classe FilaPrioridade. A base para armazenar os dados é uma lista, com as operações determinadas pela regra de prioridade. Sendo assim, é possível realizar a implementação utilizando um vetor estático, um vetor dinâmico ou uma estrutura chamada *heap*. As devidas implementações serão apresentadas nos próximos tópicos.

1 Implementando a fila de prioridades com vetores

Em Java, existem dois tipos de vetores que auxiliam a implementação das filas de prioridades. O primeiro tipo utilizado é o vetor estático, ou vetor primitivo. Ele possui algumas limitações durante a implementação, o que, em alguns casos, pode ser algo determinante na decisão sobre seu uso. A segunda possibilidade de implementação com o uso

de vetores é pela API Collections do Java. A API fornece recursos de maneira a otimizar o trabalho na implementação da fila de prioridade, além de trabalhar com alocação dinâmica de memória, facilitando o controle da quantidade de itens armazenados no ciclo do sistema.

Para melhor entendimento dos métodos básicos utilizados na estrutura de uma fila de prioridade, a primeira demonstração de implementação será feita com vetores primitivos. A implementação será realizada com a fila de prioridade sem restrições, assim são evitadas algumas verificações de limites que podem gerar confusão na compreensão das operações realizadas pelo código.

Como são necessárias duas informações base para compor um item que será armazenado na fila de prioridade, o objeto ou valor e sua prioridade, é importante criar uma classe que o represente. Essa classe é implementada de forma genérica para que, no momento do uso, seja determinado o tipo do valor que será armazenado no item.



PARA SABER MAIS

Tipos genéricos são muito utilizados em diversos sistemas e implementações de algoritmos em que os dados são dinâmicos. Para mais informações, consulte: <https://docs.oracle.com/javase/tutorial/java/generics/types.html> (acesso em: 20 abr. 2020).

Por se tratar de uma classe que representa o objeto que modela os itens armazenados na fila de prioridade, sua implementação é simples. Ela apenas define os atributos; um construtor que requisita os dois atributos, tornando-os obrigatórios; os métodos *getters*; e um método para formatar a exibição do item como texto. A implementação da classe *Item* é apresentada a seguir utilizando como premissa a técnica para tipos genéricos do Java.

```

1.  public class Item<T> {
2.      private T valor;
3.      private int prioridade;
4.
5.      public Item(T valor, int prioridade) {
6.          this.valor = valor;
7.          this.prioridade = prioridade;
8.      }
9.
10.     public T getValor() {
11.         return valor;
12.     }
13.
14.     public int getPrioridade() {
15.         return prioridade;
16.     }
17.
18.     @Override
19.     public String toString() {
20.         return String.format("(%d - %s)",
21.                                this.prioridade,
22.                                this.valor.toString());
23.     }
24. }

```

Com a classe `Item` definida, é possível construir a classe `FilaPrioridade` para compreender o funcionamento da estrutura de dados. Ela é apresentada a seguir, e na sequência a explicação linha a linha sobre o código.

```

1.  public class FilaPrioridade {
2.      private Item<?>[] itens;
3.      private int tamanho;
4.      private int capacidade;
5.
6.      public FilaPrioridade(int capacidade) {
7.          this.tamanho = 0;
8.          this.capacidade = capacidade;
9.          this.itens = new Item[this.capacidade];
10.     }
11.
12.     public FilaPrioridade() {
13.         this(10);
14.     }
15.
16.     public boolean ehVazia() {
17.         return this.tamanho == 0;
18.     }
19.
20.     public int tamanho() {
21.         return this.tamanho;
22.     }
23.
24.     public void enqueue(Item<?> i) {
25.         if(this.capacidade == this.tamanho) {
26.             throw new RuntimeException("A fila
está cheia.");
27.         }
28.
29.         this.itens[this.tamanho++] = i;
30.     }
31.
32.     public Item<?> dequeue() {
33.         if(this.ehVazia()) {
34.             throw new RuntimeException("A fila
está vazia.");
35.         }
36.
37.         int indice = this.buscaMaiorPrioridade();
38.         Item<?> altaPrioridade = this.itens[indice];

```

```

39.         for(int i = indice; i < this.tamanho() - 1;
i++) {
40.             this.itens[i] = this.itens[i + 1];
41.         }
42.
43.         this.tamanho--;
44.         return altaPrioridade;
45.     }
46.
47.     private int buscaMaiorPrioridade() {
48.         int p = 0;
49.         for (int i = 0; i < this.tamanho(); i++) {
50.             if (this.itens[p].getPrioridade() >
51.                 this.itens[i].getPrioridade())
52.                 p = i;
53.         }
54.     }
55.     return p;
56. }
57.
58. @Override
59. public String toString() {
60.     return Arrays.toString(this.itens);
61. }
62. }

```

A classe é iniciada ao declarar os atributos necessários para a manipulação e armazenamento das informações. Na linha 2, é declarado um vetor de itens, seguido por dois inteiros, o *tamanho* (linha 3), que representa a quantidade de elementos armazenados dentro fila, e a *capacidade* (linha 4), representando a quantidade máxima comportada pela fila. Logo a seguir, os dois construtores são declarados (entre as linhas 6 e 14), com a diferença de aceitar a capacidade máxima igual a 10 (segundo construtor) ou determinar a capacidade máxima no momento da instância do objeto. Da linha 16 à linha 18, o método que informa se a fila está vazia ou não é apresentado.

A implementação do método que consulta o tamanho da lista no momento corrente é realizada também (linhas 20 a 22). Ambos os métodos de suporte auxiliando as validações básicas sobre a regra para uma fila de prioridade. O próximo método implementado é o *enfileira* (linhas 24 a 30), que recebe como parâmetro um *Item*. Existe a validação para identificar se a fila não atingiu sua capacidade máxima (linhas 25 a 27), e, na sequência, o item é adicionado ao final da fila (linha 29). O método *desenfileira* é apresentado na sequência entre as linhas 32 e 45. O método deve retornar o item que está saindo da fila no momento de sua chamada. O primeiro passo importante do método é validar que existe um item para sair, caso contrário, deve-se lançar uma exceção informando que a fila está vazia (linhas 33 a 35). Validado que existem itens na fila, deve-se buscar pelo item de maior prioridade (lembrando que a maior prioridade é o 0 ou o menor inteiro entre as prioridades na fila). A busca do item de maior prioridade é realizada através do método privado *buscaMaiorPrioridade*, implementado entre as linhas 47 e 56. Ele foi implementado separado por uma questão de organização do código e para facilitar a leitura. O método *buscaMaiorPrioridade* retorna a posição da primeira ocorrência de maior prioridade na fila. Essa posição é recebida pelo método *desenfileirar* (linha 37), que imediatamente separa o item correspondente (linha 38) para retornar ao final. Entre as linhas 39 e 41 são reposicionados os itens dentro do vetor, e depois o tamanho é decrescido em um ponto (linha 43). O último método implementado para auxiliar na exibição de lista como texto é o *toString* (linhas 58 a 61).

A seguir é apresentada a classe *App*, demonstrando o uso da fila de prioridade em um simples teste. O exemplo utilizado é o mesmo do código demonstrado na introdução deste capítulo, o problema das cores.

```

1.  public class App {
2.      public static void main(String[] args) {
3.
4.          FilaPrioridade f = new FilaPrioridade(6);
5.          try {
6.              f.enfileira(new Item<String>("roxo",
7.              5));
8.              f.enfileira(new Item<String>("preto",
9.              1));
10.             f.enfileira(new
11.             Item<String>("laranja", 0));
12.             f.enfileira(new
13.             Item<String>("branco", 4));
14.             f.enfileira(new Item<String>("verde",
15.             1));
16.             f.enfileira(new
17.             Item<String>("amarelo", 3));
18.         } catch(RuntimeException e) {
19.             System.err.println(e.getMessage());
20.         }
21.         System.out.printf("%s\n\n", f);
22.
23.         try {
24.             while(!f. ehVazia()) {
25.                 System.out.println(f.
26.                 desenfileira());
27.             }
28.         } catch(RuntimeException e) {
29.             System.err.println(e.getMessage());
30.         }
31.     }
32. }

```

Na linha 4, uma fila de prioridade é declarada determinando sua capacidade igual a 6 itens. Entre as linhas 5 e 14 os itens são adicionados, informando primeiro o texto representando o valor do item na fila e na sequência a prioridade do item. A fila de prioridade completa é exibida na linha 15. Depois, entre as linhas 17 e 23, é realizado um laço de repetição do tipo *while* para desenfileirar os itens enquanto a fila não estiver vazia. O resultado desse código é apresentado a seguir.

```
[(5 - roxo), (1 - preto), (0 - laranja), (4 - branco), (1  
- verde), (3 - amarelo)]
```

```
(0 - laranja)  
(1 - preto)  
(1 - verde)  
(3 - amarelo)  
(4 - branco)  
(5 - roxo)
```

1.1 Trabalhando com o vetor dinâmico

A fila de prioridade implementada possui alguns pontos que dificultam o controle dos objetos. Além de seu tamanho ser limitado, muitas operações executadas são grandes e podem gerar problemas de compreensão e leitura do código-fonte. Para auxiliar uma implementação mais enxuta e com mais recursos de melhor desempenho, é utilizada a classe `ArrayList` da API Collections do Java. Ela é mais dinâmica e mais confiável para implementar uma fila de prioridade utilizando vetores. A implementação é apresentada a seguir.

```
1. public class FilaPrioridade {  
2.     private List<Item<?>> itens = new  
   ArrayList<Item<?>>();  
3.  
4.     public boolean ehVazia() {  
5.         return this.itens.isEmpty();  
6.     }  
7.  
8.     public int tamanho() {
```

```

9.         return this.itens.size();
10.    }
11.
12.    public void enfileira(Item<?> i) {
13.        this.itens.add(i);
14.    }
15.
16.    public Item<?> desenfileira() {
17.        if(this.ehVazia()) {
18.            throw new RuntimeException("A fila
está vazia.");
19.        }
20.
21.        Item<?> altaPrioridade = this.itens.get(0);
22.        for(Item<?> i : this.itens) {
23.            if(i.getPrioridade() <
altaPrioridade.getPrioridade()) {
24.                altaPrioridade = i;
25.            }
26.        }
27.
28.        this.itens.remove(altaPrioridade);
29.        return altaPrioridade;
30.    }
31.
32.    @Override
33.    public String toString() {
34.        return this.itens.toString();
35.    }
36. }

```

Utilizar a API Collections facilita a implementação em muitos pontos. A quantidade de linhas de código entre ambas as classes apresentadas diminui em aproximadamente um terço. Apenas um atributo é necessário ser declarado, o vetor de itens que é um `ArrayList`. Os métodos *ehVazia* (linhas 4 a 6) e *tamanho* (linhas 8 a 10) foram simplificados em seu corpo. Eles utilizam métodos implementados através da classe `ArrayList`. O método *enfileira* ficou ainda mais simples, uma vez que não existe a necessidade de se validar se capacidade máxima foi ou não atingida

(linhas 12 a 14). Desenfileirar (linhas 16 a 30) ficou mais fácil, uma vez que é possível passar a referência do objeto que deve ser removido do vetor. Então é necessário apenas percorrer o vetor em busca do item de maior prioridade. Caso existam dois com a mesma prioridade, a ordem de inserção prevalece como maior prioridade. Por fim, o método *toString* também foi simplificado para exibir melhor o vetor como um texto.

Perceba que o método *desenfileira* ficou com o maior trabalho (principalmente na implementação com vetor primitivo), pois ele deve identificar qual é o item de maior prioridade para removê-lo. Mas essa é a estratégia que facilita na hora de armazenar e ela apresenta um desempenho menor na remoção do item. Uma outra estratégia é determinar a posição de enfileiramento no momento da inclusão do item na fila. Assim, o gasto de tempo será menor quando ele for removido. Entretanto, a melhor opção para implementar uma fila de prioridade é utilizando uma estrutura de dados do tipo heap. Essa estrutura será apresentada no tópico a seguir.

2 Heap

As implementações feitas até o momento na fila de prioridade não possuem um desempenho ótimo, o que significa que podem ser melhoradas. O principal ganho que pode existir nas operações de uma fila de prioridade são a inserção ou remoção de um item, que têm em seu pior caso uma complexidade $O(n)$. A ideia para melhorar essas operações básicas é criar um balanceamento entre a inserção e a remoção das informações. Assim, não existe um pior caso apenas em uma ou em outra, ambas conseguem manter um bom desempenho (GOODRICH; TAMASSIA, 2013).

Para balancear a implementação de uma fila de prioridade, utiliza-se uma estrutura de dados chamada heap. O tempo das operações de inserção e remoção de dados em um heap é de $O(\log(n))$. Esse é um excelente resultado quando comparado com as demais implementações baseadas em sequência. Uma estrutura do tipo heap tem como premissa

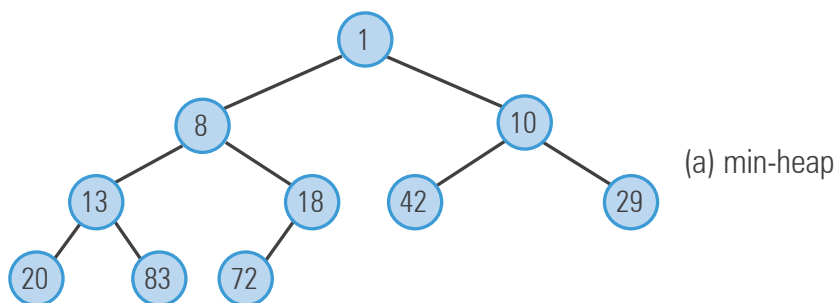
o abandono do uso de uma lista sequencial na maneira de armazenar e buscar uma informação, e como base uma árvore binária (CORMEN et al., 2009; GOODRICH; TAMASSIA, 2013).

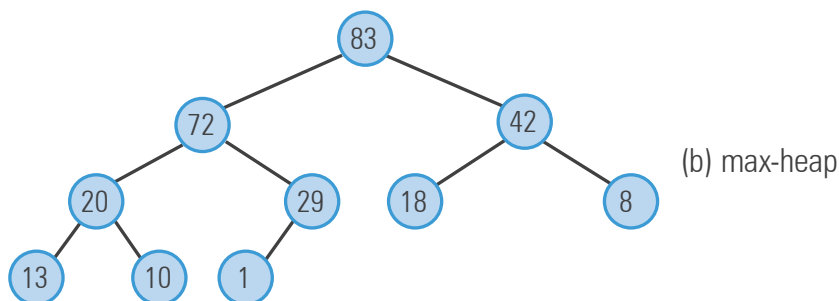
A árvore binária que representa o heap (A) armazena uma coleção de nós internos. Os nós podem ser compostos por chave e valor ou apenas valor, o importante é que exista uma maneira de estabelecer uma ordem de comparação entre eles. Além disso, o heap A deve manter as seguintes propriedades (GOODRICH; TAMASSIA, 2013):

- **relacional:** que define a forma como os nós serão armazenados; e
- **estrutural:** que define a estrutura com base nos próprios nós armazenados.

A ordem dos nós na estrutura heap é dada de duas maneiras: max-heap e min-heap. O max-heap tem como propriedade para ordenação que os nós filhos são sempre menores que o seu nó pai. É importante ressaltar que, pela regra de uma árvore binária (que compõe o heap), um nó só pode gerar dois filhos. O min-heap tem a propriedade de ordenação que os nós pais são sempre menores que seus filhos (NECAISE, 2010). A figura 2 apresenta um exemplo de um heap aplicando as duas propriedades, max-heap e min-heap.

Figura 2 – Duas possíveis ordenações do heap apresentadas: (a) min-heap priorizando o menor elemento; (b) max-heap priorizando o maior elemento





A figura 2 (a) apresenta a formação de um heap, no formato de árvore binária, optando pela ordenação do min-heap, na qual os filhos são maiores que seus pais. Enquanto isso, na figura 2 (b) é apresentada a ordenação pela propriedade max-heap, na qual os filhos são menores que seus pais. É importante ressaltar alguns termos que são oriundos da árvore binária e utilizados também no formato heap. O preenchimento de uma árvore binária sempre é realizado da esquerda para a direita. Os nós 1 (min-heap) e 83 (max-heap) são denominados raízes do heap. E os nós que não possuem filhos até o momento são chamados de nós folhas (TENENBAUM; LANGSAM; AUGENSTEIN, 2004; CORMEN *et al.*, 2009).

O heap é uma estrutura de dados especializada que possui operações limitadas. A inserção de um novo nó dentro de um heap deixa-o posicionado de acordo com a regra de ordenação adotada. A remoção só é realizada através da raiz do heap. Para continuar os estudos dessa estrutura de dados, os exemplos serão guiados pela propriedade do max-heap. O min-heap pode ser obtido por meio da inversão dos condicionais apresentados no algoritmo.

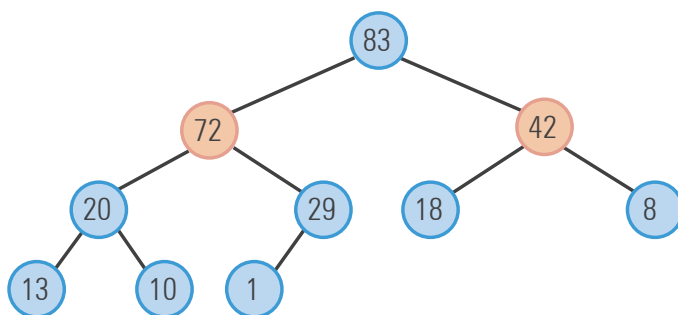
2.1 Inserindo um novo nó

Quando qualquer novo nó for inserido dentro do heap, as propriedades de ordem e formato devem ser mantidas. Sendo assim, identificar

onde o novo nó será inserido é uma pequena parte do problema. A maior questão é como fazer a inserção se a posição em que esse nó deve permanecer está ocupada por um outro nó. Sendo assim, a probabilidade de ocorrer uma movimentação entre os nós é extremamente alta. Dessa forma, o nó deve ser inserido sempre pelas folhas e não pela raiz (que é o sentido de leitura padrão). Antes de inserir por uma folha, deve-se procurar pelo pai que possui apenas um filho. Ele deve ser priorizado (NECAISE, 2010).

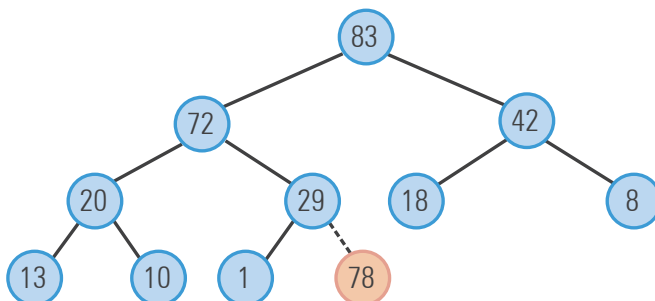
O primeiro passo do algoritmo é identificar as posições que podem ser ocupadas pelo novo nó. Para exemplificar melhor, o max-heap apresentado na figura 2 será utilizado até o final do capítulo. O nó de chave 78 será inserido no heap. A figura apresenta em laranja as duas possíveis posições que o nó 78 pode ocupar após o processo de inserção.

Figura 3 – Em laranja são apresentadas as possíveis posições que o nó 78 pode ocupar depois do processo de inserção



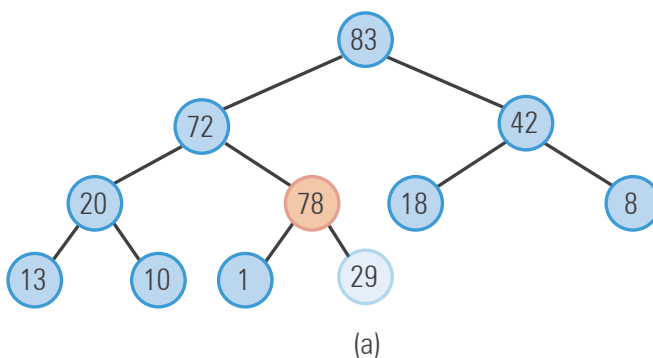
Na sequência, o nó é criado e anexado com uma folha, no exemplo, ele é uma folha, filha do nó 29, que possui apenas um filho. Veja na figura 4 o nó 78 sendo anexado como uma folha (elemento na cor laranja).

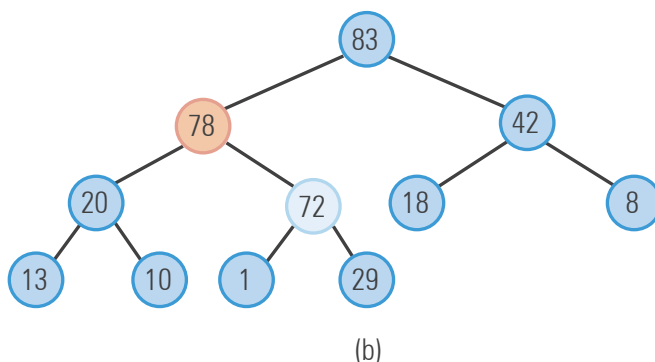
Figura 4 – Nó 78 (em laranja) é criado e anexado como uma folha, filha de 29



Nesse momento, a propriedade max-heap de ordenação está comprometida. Para restaurar a propriedade de ordenação do heap, o nó inserido deve percorrer o caminho reverso do heap, em direção à raiz, até encontrar a posição correta. Essa operação é chamada de *sift-up*, que também é conhecida como *up-heap*, *heapify-up*, *bubble-up*, entre outros nomes (NECAISE, 2010). A figura 5 (a) mostra o primeiro passo da operação *sift-up*, na qual existe uma troca entre os nós 29 (cor mais clara) e 78 (em laranja). A figura 5 (b) exibe o segundo e último passos, para o exemplo, nos quais a troca do nó 78 (em laranja) é feita com o nó 72 (cor mais clara), reestabelecendo a ordem do max-heap.

Figura 5 – Apresentação do processo de *sift-up* reestabelecendo a ordem max-heap





Com o balanceamento realizado, a operação para a inserção do nó de chave 78 foi concluída. Na próxima seção, é apresentado o processo da operação de remoção do nó raiz da estrutura de dados heap.

2.2 Removendo um nó

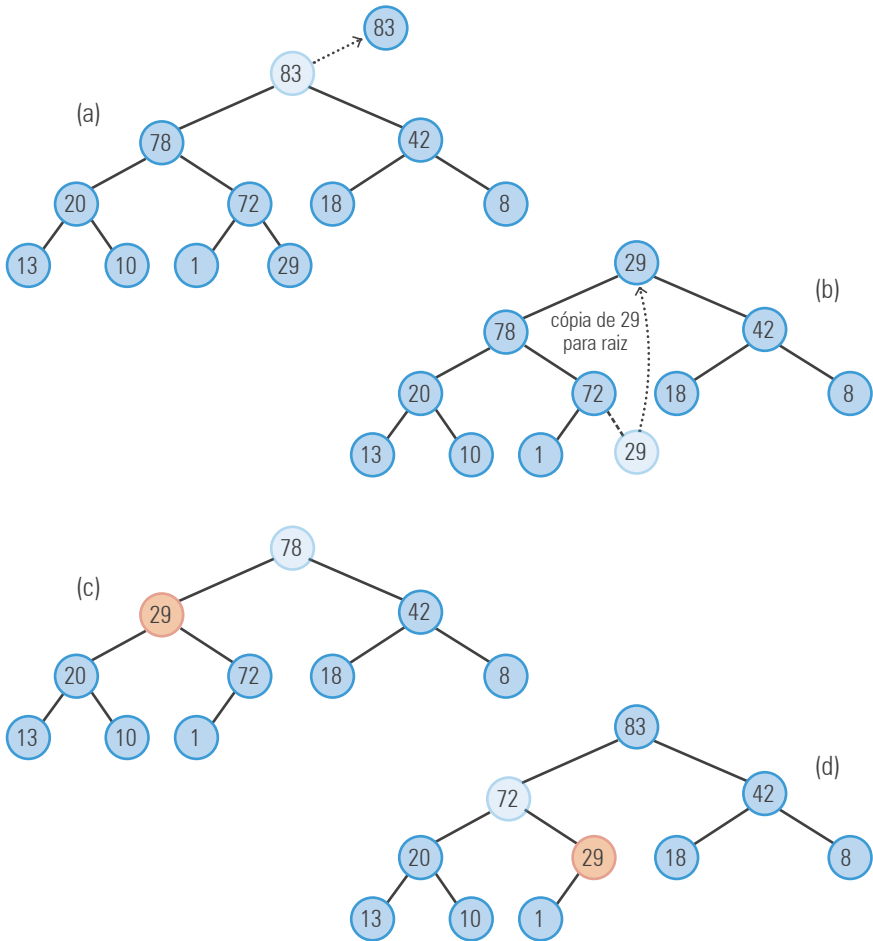
A extração de um nó dentro de uma estrutura de heap só pode ocorrer através do nó raiz. Sendo assim, se a ordenação for realizada através da regra max-heap, sempre serão removidos os nós maiores primeiro. No caso de se optar por uma implementação de min-heap, os nós menores terão prioridade na remoção (NECAISE, 2010; GOODRICH; TAMASSIA, 2013).

Para que a ordem seja mantida após a remoção de um nó, um nó folha deve assumir o lugar da posição do nó raiz. Apesar de esse processo de substituição ser claro e de fácil implementação, ainda não garante por completo todas as propriedades necessárias para manter a ordem do heap (NECAISE, 2010). A folha escolhida, pela regra da árvore binária, é a mais à direita da árvore.

Após esse processo de cópia, o nó que foi copiado para a raiz deverá percorrer o caminho da árvore até que ela se torne novamente uma folha. Quando o processo for finalizado, a estrutura heap novamente

estará organizada e ordenada. O processo é conhecido como *sift-down* e é apresentado na figura 6.

Figura 6 – Processo para remoção do nó raiz 83 junto com o *sift-down* aplicado ao nó 29, após seu reposicionamento no heap



No processo de remoção apresentado pela figura 6, é possível identificar os quatro passos executados para manter as propriedades de um heap. No passo (a), o nó 83, que era a raiz, é removido do heap. Para reestabelecer a ordem e o formato do heap, a última folha à direita é

reposicionada como a nova raiz do heap, conforme o passo (b). Nesse momento, o processo *sift-down* é inicializado e o nó raiz corrente (chave 29, em laranja) é trocado de posição com o 78 (cor mais clara). O primeiro passo (c) do *sift-down* é realizado e, em sequência, o segundo passo (d). Neste último, existe a troca entre os nós de chave 72 (cor mais clara) e 29 (em laranja). O heap, agora, está balanceado novamente.

2.3 Implementando um heap

Com as operações definidas, é hora de implementar a estrutura de heap apresentada com a propriedade de max-heap. Confira a seguir a implementação da classe Heap em Java.

```
1. public class Heap {
2.     private List<Item<?>> itens = new
   ArrayList<Item<?>>();
3.
4.     public int tamanho() {
5.         return this.itens.size();
6.     }
7.
8.     public boolean ehVazia() {
9.         return this.itens.isEmpty();
10.    }
11.
12.    public void adiciona(Item<?> i) {
13.        this.itens.add(i);
14.        this.siftUp(this.itens.size() - 1);
15.    }
16.
17.    public Item<?> remove() {
18.        Item<?> removido = this.itens.get(0);
19.        this.itens.set(0, this.itens.get(this.itens.
size() - 1));
20.        this.itens.remove(this.itens.size() - 1);
21.        this.siftDown(0);
22.        return removido;
}
```

```

23.     }
24.
25.     private void siftUp(int n) {
26.         if (n > 0) {
27.             int pai = Math.floorDiv(n, 2);
28.             if(this.itens.get(n).getPrioridade()
29. >
30.                 this.itens.get(pai).
31. getPrioridade()) {
32.                 this.swap(n, pai);
33.                 this.siftUp(pai);
34.             }
35.         }
36.     private void siftDown(int n) {
37.         int esquerda = 2 * n + 1;
38.         int direita = 2 * n + 2;
39.         int maior = n;
40.
41.         if((esquerda < this.itens.size()) &&
42.             this.itens.get(esquerda).
43. getPrioridade() >=
44.                 this.itens.get(maior).
45. getPrioridade()) {
46.             maior = esquerda;
47.         } else if((direita < this.itens.size()) &&
48.             this.itens.get(direita).
49. getPrioridade() >=
50.                 this.itens.get(maior).
51. getPrioridade()) {
52.             maior = direita;
53.         }
54.         if(maior != n) {
55.             this.swap(n, maior);
56.             this.siftDown(maior);
57.         }
58.     }

```

```

58.     private void swap(int a, int b) {
59.         Item<?> temp = this.itens.get(a);
60.         this.itens.set(a, this.itens.get(b));
61.         this.itens.set(b, temp);
62.     }
63.
64.     @Override
65.     public String toString() {
66.         return this.itens.toString();
67.     }
68. }

```

No código apresentado, é utilizado um `ArrayList` para armazenar os itens da estrutura heap. Apesar de ser utilizada uma lista, em sua essência, a forma de percorrer é diferente, ou seja, não é mais linear. A forma de percorrer a lista caracteriza essa estrutura de dados como uma árvore binária. Os métodos de suporte *tamanho* (linhas 4 a 6) e *ehVazia* (linhas 8 a 10) são os primeiros implementados na classe `Heap`. Na sequência, é apresentada a implementação do método *adiciona* (linhas 12 a 15), que recebe como parâmetro um `Item`, o mesmo tipo definido no início do capítulo para manter o contexto da fila de prioridade. Após adicionar o item no `ArrayList` (linha 13), o método *siftUp* é invocado, recebendo o último índice da lista como parâmetro. O método *siftUp* (linhas 25 a 34) verifica se o índice recebido não se refere ao índice da raiz (linha 26), depois identifica o índice do nó pai, que é o número inteiro da divisão do índice recebido por 2 (linha 27). Com o pai determinado, verifica-se se o nó tem prioridade maior que a do nó pai. Em caso positivo, é feita uma troca de posições com o auxílio do método *swap* (linhas 58 a 62). O índice do nó pai é o alvo agora, então ele é enviado em uma chamada recursiva do método *siftUp* até todos os nós terem sido percorridos.

O próximo método é o método *remove* (linhas 17 a 23), que faz uma cópia do item a ser removido (linha 18), copiando a última folha para a posição da raiz (linha 19). Na sequência, o método *siftDown* é invocado, recebendo 0 (o índice da raiz) como parâmetro. No método *siftDown*

(linhas 36 a 56), são selecionados os filhos da esquerda (linha 37) e da direita (linha 38) e o nó recebido é identificado como o maior (linha 39). Na sequência, os nós da esquerda e da direita são comparados e verificados se são maiores que os nó pai (linhas 41 a 49). Em caso positivo, eles passam a ser o índice do maior elemento e é feita uma troca com o pai (linha 52). Depois, o método *siftDown* é chamado de maneira recursiva para continuar a ordenação do heap.

Apesar de a implementação seguir a propriedade de max-heap, para utilizar o heap como uma fila de prioridade basta implementá-lo com a propriedade min-heap.

3 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Implemente o heap para uma fila de prioridade.
2. Execute o teste para verificar o desempenho de cada uma das estruturas. Faça uma tabela comparativa.
3. Implemente um sistema de emissão de passagens aéreas em um terminal com regras de prioridades para idosos, gestantes e adultos acompanhados de crianças. Ao todo, devem ser 4 níveis de prioridade, pelo menos.

Considerações finais

Apresentamos ao longo deste capítulo os conceitos de fila de prioridade e heap e de como eles se relacionam. Eles podem ser utilizados em diversas aplicações que envolvam classificações de prioridade, como o caso de sistemas de mensageria e até em filas de bancos, supermercados e aeroportos, onde existam regras de prioridades ou preferenciais. Algumas implementações não são tão triviais e até o resultado prático

pode divergir um pouco do teórico, já que a divisão visual é mais intuitiva do que a divisão da árvore através do código. São muitos os arredondamentos que podem divergir um pouco do visual. Contudo, isso não quer dizer que haja algum erro, as propriedades são e sempre devem ser mantidas durante todo o processo. Alinhando o conhecimento da estrutura de dados com os algoritmos, muitas possibilidades devem ser exploradas para otimizações e soluções de novos problemas.

Referências

CORMEN, Thomas H. *et al.* **Introduction to algorithms**. Cambridge: MIT Press, 2009.

DEITEL, Paul J.; DEITEL, Harvey M. **Java**: como programar. São Paulo: Pearson, 2008.

FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Prentice Hall, 2005.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

NECAISE, Rance D. **Data structures and algorithms using Python**. Hoboken: John Wiley & Sons, Inc., 2010.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: MAKRON Books, 2004.

Sobre o autor

Andrey Araujo Masiero é doutor em engenharia elétrica na área de inteligência artificial pelo Centro Universitário da Fundação Educacional Inaciana Padre Sabóia de Medeiros (FEI) e bolsista PROSUP/CAPES. Mestre em engenharia elétrica na área de inteligência artificial pelo Centro Universitário FEI e bolsista em tempo integral do projeto FINEP Pesquisa e Estatística baseada em Acervo Digital de Prontuário Médico do Paciente em Telemedicina Centrada no Usuário. Graduado em ciência da computação pelo Centro Universitário FEI em 2009. Premiado com o primeiro lugar na Expocom com o trabalho Sistema de Gerenciamento de Patterns, Anti-Patterns e Personas (SIGEPAPP). Profissional com 16 anos de experiência no mercado, participou de projetos com o governo do Estado de São Paulo e em projetos da sociedade privada, como a integração entre os bancos Santander e Real. Áreas de interesse: engenharia de usabilidade, interação homem-computador, interação humano-robô, engenharia de software, inteligência artificial, aprendizado de máquina, data mining e algoritmos computacionais.

