



ESTRUTURAS DE PADOS

VINÍCIUS GODOY

Estruturas de dados

Vinícius Godoy

© 2020 – IESDE BRASIL S/A.

É proibida a reprodução, mesmo parcial, por qualquer processo, sem autorização por escrito do autor e do detentor dos direitos autorais.

Projeto de capa: IESDE BRASIL S/A.

Imagen da capa: carlos castilla/Shutterstock

CIP-BRASIL. CATALOGAÇÃO NA PUBLICAÇÃO
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

G535e

Godoy, Vinícius

Estruturas de dados / Vinícius Godoy. - 1. ed. - Curitiba [PR] : IESDE, 2020.

162 p. : il.

Inclui bibliografia

ISBN 978-85-387-6573-8

1. Estrutura de dados (Programação). 2. Programação (Computadores). I. Título.

19-61226

CDD: 005.1

CDU: 004.42

Todos os direitos reservados.



IESDE BRASIL S/A.

Al. Dr. Carlos de Carvalho, 1.482. CEP: 80730-200
Batel – Curitiba – PR
0800 708 88 88 – www.iesde.com.br

Vinícius Godoy

Mestre em Visão Computacional pela Pontifícia Universidade Católica do Paraná (PUCPR), especialista em Desenvolvimento de Jogos de Computadores pela Universidade Positivo (UP) e graduado em Tecnologia em Informática pela Universidade Tecnológica Federal do Paraná (UTFPR). Trabalha na área de informática desde 1997, tendo participado de grandes projetos, como a edição de 100 anos do Dicionário Aurélio Eletrônico. Também atua como moderador do GUJ, o maior fórum de tecnologia Java do Brasil.

Videoaula em QR code!

Agora é possível acessar nossas videoaulas por meio dos QR codes inseridos no livro.

Note que existe, ao lado do início de cada seção de capítulo, um QR code (código de barras) para acessar a videoaula.

Para acessá-la automaticamente, basta direcionar a câmera fotográfica de seu smartphone, tablet ou notebook para o QR code.

Em algumas versões de smartphone, é necessário ter instalado um aplicativo para ler QR code, disponível gratuitamente na App Store e na Google Play.



SUMÁRIO

1 A informação no computador 9

- 1.1 A memória 9
- 1.2 Tipos primitivos 14
- 1.3 Vetores 15
- 1.4 Classes 16

2 Pilhas e filas 22

- 2.1 Conceitos 22
- 2.2 Implementação da pilha 25
- 2.3 Implementação da fila 36

3 Listas 46

- 3.1 Conceitos 46
- 3.2 Implementação da lista estática 48
- 3.3 Implementação da lista encadeada 61
- 3.4 Últimos ajustes 72

4 Ordenação de dados 84

- 4.1 Conceitos 84
- 4.2 Algoritmos de ordenação 86
- 4.3 Listas ordenadas 98

5 Espalhamento 106

- 5.1 Conceitos 106
- 5.2 Implementação do mapa não ordenado 117

6 Árvores binárias 134

- 6.1 Conceitos 134
- 6.2 Implementando um mapa ordenado 136
- 6.3 Outros tipos de árvore 151

APRESENTAÇÃO

Informática vem do francês *information automatique* (informação automática). No passado, também foi chamada de processamento de dados. Não é mera coincidência que o nome da área de informática sempre se refira a dados e à informação.

Hoje os computadores são responsáveis por processar um volume enorme de dados com extrema rapidez. Entretanto, por mais rápidos que se tornem, é importante que o programador utilize bons algoritmos, capazes de gerenciar todos os recursos com a máxima eficiência.

Ao iniciar um projeto, seja o sistema de um grande banco ou um aplicativo para o celular, várias questões surgem, como: qual é o volume de dados que será processado? O acesso a esses dados ocorre em sequência? Há alguma forma de ordenar esses dados? As informações são independentes ou estão relacionadas? Neste livro, você começará a responder a essas perguntas conhecendo o conceito das estruturas de dados.

No primeiro capítulo, entendemos como o computador armazena a informação, revisando os tipos de memória existentes e lembrando de como os dados são representados em seu interior, de acordo com seu tipo. No segundo capítulo, estudamos duas estruturas de dados simples: as pilhas e as filas, que, mesmo permitindo apenas acesso controlado à informação, já possuem grande utilidade prática. Já no terceiro capítulo, discutimos as listas, um tipo flexível de estrutura que está presente na maioria das aplicações, substituindo os vetores na maior parte dos programas.

No quarto capítulo, estudamos o conceito de *ordenação de dados*. Para tanto, analisamos diferentes formas de ordenar vetores e suas características, além de entender como o computador pode usar o conceito de ordenação para realizar buscas de valores de maneira extremamente eficiente.

No quinto capítulo, aprofundamos nosso estudo por meio das tabelas hash, que combinam vetores e listas para criar uma poderosa estrutura capaz de localizar rapidamente objetos não ordenados por meio de uma estratégia conhecida como espalhamento. Isso nos permite implementar os conceitos dos mapas e dos conjuntos não ordenados. Por fim, no sexto e último capítulo, conhecemos as árvores binárias, que mantêm os dados na memória de maneira naturalmente ordenada.

Para tudo isso, utilizamos a linguagem de programação Java, que você já deve conhecer. Por ser multiplataforma, você poderá utilizá-la em seu sistema operacional favorito, além de se beneficiar de seus editores de códigos poderosos e gratuitos, de sua preferência.

Esperamos que você aproveite ao máximo este livro. Ele foi elaborado com muito cuidado, pensando em sua formação.

Tenha uma excelente leitura!

A informação no computador

O computador é binário. Isso significa que toda informação em seu interior, até mesmo as instruções dos programas que ele executará, será codificada como uma sequência de vários 0 e 1 intercalados. Quem realiza a interpretação de toda essa informação é o processador. Nele, está embutido um conjunto básico de comandos que permitem que sejam feitas comparações lógicas, somas, manipulação de dados na memória, desvios condicionais, entre outros.

Pode parecer incrível, mas é da movimentação de dados de uma região a outra da memória e do processamento de algumas operações matemáticas simples que todos os programas são feitos, sejam planilhas eletrônicas, jogos ou mesmo a linguagem de programação.

É por isso que, para se escrever um software eficiente, é importante compreender bem as características de funcionamento tanto da memória quanto do processador, e como os computadores representam a informação em seu interior. Assim, esse será o foco deste capítulo.

1.1

A memória



Toda a informação do computador está presente na memória. Porém, ela não é um único componente, uniforme. A memória pode variar de maneira tanto física, estando presente em dispositivos diferentes, quanto lógica, em áreas organizadas pelo sistema operacional. Desse modo, a seguir, vamos estudar essas duas formas.



1.1.1 Dispositivos de memória

Existe uma série de dispositivos responsáveis por armazenar informação no computador, que variam em diversas características, como

a abundância de informação que são capazes de armazenar, a velocidade de acesso, a durabilidade e o custo. Via de regra, podemos traçar uma relação direta entre custo e velocidade: quanto mais rápida a memória é, mais cara ela também será – e isso quase sempre será levado em consideração na construção de um hardware.

De acordo com Tanenbaum e Bos (2016), todos os dispositivos são colocados em duas grandes categorias: os voláteis e os não voláteis. A grande característica do **armazenamento volátil** é que a informação em seu interior se perde quando o computador é desligado. O equipamento que a armazena, por meio de processos eletrônicos extremamente velozes, é chamado de **pente de memória**.

Um único computador tem diferentes tipos de memória volátil:

MEMÓRIA CACHE

Pouco abundante, mas muitíssimo veloz. Encontra-se fisicamente perto do processador, quando não está ligada diretamente a ele. Essa proximidade evita a perda de velocidade no transporte da informação.

MEMÓRIA RAM

Mais abundante, porém mais lenta que a memória cache. Nela, estão contidos os programas e os dados.

Quando uma instrução é realizada no processador, este buscará os dados na memória cache primeiro. A memória RAM só será consultada se ele não os encontrar lá. Nesse caso, a informação é utilizada e trazida para o cache. Dessa forma, quanto mais frequente for o acesso à mesma informação, mais beneficiada pelo cache a aplicação será.

Obviamente, os computadores seriam pouco úteis se a informação sempre se perdesse. Para isso, o **armazenamento e os dispositivos não voláteis** foram criados. Eles são capazes de armazenar a informação por meses, ou até anos, sendo alguns exemplos:



Discos rígidos (HDs)

São discos magnéticos que funcionam imantando partes de sua extensão através de uma agulha presa a um braço. Devido a essa característica, eles leem informações sequencialmente, o que é muito mais rápido do que acessar informações aleatórias, pois estas se encontram espalhadas no disco.

Discos de estado sólido (SSD)

São dispositivos sem partes móveis, mais rápidos e silenciosos que os tradicionais discos rígidos e de menor capacidade (os maiores têm em torno de 2 tb). São também consideravelmente mais caros.



Memória flash (pendrive)

São pentes de memória não voláteis. Costumam ser pequenos, portáteis e ligados ao computador por meio de uma porta USB. Os pendrives tornaram o disquete obsoleto por serem portáteis, silenciosos e confiáveis, além de terem milhares de vezes mais capacidade de armazenamento.

Fitas de dados (DAT)

São fitas magnéticas de baixo custo de armazenamento. Sua grande característica é a resiliência, ou seja, se corretamente armazenadas, duram muitos anos. Por isso, ainda são muito usadas para backup. Sua velocidade de acesso é muito baixa e permite praticamente somente o acesso sequencial.



Outras mídias

Várias outras mídias se popularizaram e caíram (ou estão caindo) em desuso com os anos, como os disquetes, CDs e DVDs.

É importante destacar o papel dos dois primeiros dispositivos. Tanto os discos SSD quanto os rígidos são utilizados pelo sistema operacional para implementar o conceito de **memória virtual**. A memória RAM é rápida e grande o suficiente para permitir que o computador trabalhe com os processos ativos. Porém, ela dificilmente seria capaz de manter sozinha a informação de todos os programas que um usuário resolva abrir de uma única vez. Por isso, o sistema operacional pode movimentar partes das informações da memória RAM para os discos, dando a ilusão de que a RAM é muito maior do que realmente é.

É importante saber que existem objetivos distintos entre os dispositivos de armazenamento volátil e os não voláteis; nestes, por exemplo, estamos interessados em maximizar a quantidade de dados que gostaríamos de armazenar. Por mais rápido que sejam, seu tempo de acesso inviabiliza o uso desse tipo de memória para processamento de informações, já que deixaria o processador ocioso, esperando a leitura dos dados.

1.1.2 Áreas de memória de um processo

A memória RAM é muito utilizada para trabalho. Quando uma aplicação é executada pelo computador, o sistema operacional a carrega e distribui seus dados em áreas distintas da memória. Essa organização também impacta a velocidade com que os dados são acessados e como os gerenciamos em linguagens de programação. Segundo Tanenbaum e Bos (2016), duas áreas são especialmente importantes na programação: a pilha (stack) e o heap.

Quando um programa é executado, suas instruções são carregadas em uma área especial de memória conhecida como *somente leitura* (read only). Além das instruções, essa área contém os valores de dados e textos literais, ou seja, diretamente informados no código, geralmente para a inicialização de variáveis ou impressão de texto. Como o próprio nome diz, os valores presentes nessa área de memória não podem ser alterados. Os projetistas têm a preocupação de tornar seu uso seguro, sem que você sequer perceba que essa área de memória existe (SIERRA; BATES, 2010). Outras linguagens de mais baixo nível, como o C e o C++, permitem que você tente alterar dados criados nesse local, o que ocasiona um erro no tempo de execução (MAIN; SAVITCH, 2005).

Como você já aprendeu em suas aulas de programação, todo programa de computador executa sequencialmente um conjunto de operações. Porém, o código pode ser subdividido em diferentes funções, cada uma com seu escopo local e suas próprias variáveis.

Quando um código que vinha sendo executado de maneira linear atinge uma chamada de função, o sistema precisa anotar a posição em que isso ocorreu, criar as variáveis locais da função que está sendo chamada e desviar a execução do programa para as instruções que lá se encontram. Da mesma forma, quando a função atinge seu término, a memória dessas variáveis deve ser eliminada e o programa precisa retornar ao ponto em que estava antes da função ser chamada.

Essa informação é armazenada numa área especial de memória chamada **pilha de execução**. Por tratar de operações comuns, essa área é organizada de modo que a criação e remoção das variáveis locais seja muito rápida e automática (TANENBAUM; BOS, 2016).

Já o **heap** representa toda a memória disponível no sistema. Essa memória precisa ser explicitamente requisitada (alocada) e liberada (desalocada) pelo programador, por meio de comandos específicos. Na linguagem C, isso é feito com as funções `malloc` e `free`; e, no C++, seus equivalentes são os comandos `new` e `delete` (MAIN; SAVITCH, 2005).

É importante entender que essas operações são custosas se comparadas ao que ocorre na pilha, porque, quando uma nova alocação é feita, o sistema precisa buscar um espaço livre correspondente em toda extensão da RAM e reservá-la ao programador. Por fim, a informação pode ficar dispersa na memória, tornando seu acesso também mais lento.

Algumas linguagens, como o Java, possuem um sistema especial para realizar automaticamente as desalocações no heap, chamado **garbage collector** (GOODRICH; TAMASSIA, 2013). Esse sistema é capaz de identificar automaticamente um objeto sem referências e eliminá-lo, acabando com a necessidade de comandos como `free` ou `delete`.

Além disso, o garbage collector realiza uma gerência inteligente no processo de alocação e desalocação, sendo capaz de, conforme Goetz (2003):

- agrupar dados que precisem ser desalocados em blocos grandes, realizando a operação de uma vez só;



CURIOSIDADE

O nome em inglês da pilha de execução é call stack e já recebeu várias outras traduções para o português, como *pilha de alocação* ou *pilha de memória*. A tradução literal seria *pilha de chamada*.



ATIVIDADE 1

Monte uma tabela que diferencie as memórias heap e stack quanto a organização, custo de alocação e disponibilidade.

- reutilizar áreas de memória recém-desalocadas em novas alocações, evitando o custo do sistema operacional em ambas as operações;
- utilizar vários núcleos de processamento para máxima performance.

 Artigo

<https://www.ibm.com/developerworks/library/j-jtp11253/index.html>

Selecionamos para você esse artigo escrito por Brian Goetz, um dos principais projetistas e criadores da linguagem Java. Nesse material, de sua coluna *Java theory and practice*, da IBM (International Business Machines Corporation), Goetz descreve o funcionamento do garbage collector em detalhes. O artigo está em inglês, mas você pode utilizar o recurso de tradução do seu navegador se necessário.

Acesso em: 24 out. 2019.

 Artigo

<https://www.ibm.com/developerworks/library/j-jtp01274/index.html>

Esse outro artigo, da mesma coluna, descreve características da performance do garbage collector. Logicamente, ele já evoluiu bastante, mas boa parte do que está descrito ainda se mantém. Se necessário, você pode usar o recurso de tradução do seu navegador.

Acesso em: 24 out. 2019.

Saiba mais

Por não conseguir controlar estes fatores da execução, o contrato da plataforma Java até exime a Oracle de responsabilidade caso seja usada em sistemas críticos, como usinas atômicas.

Como desvantagem desse sistema está a imprevisibilidade. Apesar de ser um sistema crítico e, portanto, extremamente eficiente, o programador não tem como controlar quando a execução ocorrerá, não tem como prever quanto tempo levará e nem quanta memória será desalocada. Por isso, o Java não costuma ser recomendado em aplicações com exigência de tempo real, como jogos ou **vídeo**.

1.2

Tipos primitivos



A interpretação dos dados dentro das variáveis, que representam a memória, muda de acordo com o tipo de dado. Todas as linguagens fornecem um conjunto de **tipos primitivos**, isto é, formas-padrão de se interpretar dados. Embora esses tipos variem de uma linguagem para outra, o conjunto básico geralmente inclui tipos numéricos, caracteres de texto e variáveis booleanas (verdadeiro/falso).

Conforme a Oracle (2019), no caso do Java, os tipos primitivos são chamados de:

- booleano (verdadeiro/falso): boolean;
- numéricos inteiros: byte, short, int e long;
- numéricos decimais: float e double;
- caracteres: char.

Perceba que todos os dados ainda serão um conjunto de bits para a memória. O que cada tipo de dado define é como esse conjunto será analisado e interpretado. Por exemplo, no Java, uma variável do tipo char é formada por 2 bytes e dentro contém um número. Porém, ao imprimi-la, o Java sabe que, por se tratar de um caractere, esse número deve ser mapeado a um caractere presente na tabela **Unicode**.

Além dos valores, as linguagens permitem um conjunto de operações sobre cada tipo de dado, além de regras de conversão. Por exemplo, na linguagem C, o tipo booleano não existe (MAIN; SAVITCH, 2005). Essa linguagem define que o tipo inteiro pode ser usado em seu lugar, assim, uma variável com valor 0 será considerada falsa, enquanto outros valores serão considerados verdadeiros. No Java, há um tipo de dado específico para os valores booleanos, não sendo possível usar, por exemplo, o operador de negação (!) sobre um número inteiro.

Por fim, vale ressaltar que tipos primitivos geralmente trabalham por valor, isto é, eles representam apelidos para endereços de memória. Assim, duas variáveis diferentes representarão áreas diferentes da memória, e atribuir o valor de uma a outra gerará a cópia desse valor. Note que todas essas características estão diretamente ligadas ao fato de os tipos primitivos terem sido pensados como o padrão para as variáveis locais, criadas na memória stack.

[www.](#) Site

A tabela Unicode é um padrão oficial, definido por um grupo de engenheiros. Você pode consultá-la em vários sites, como na seguinte referência:
<https://unicode-table.com/pt/>.
Acesso em: 25 out. 2019.

1.3 Vetores



As necessidades de programação exigem a manipulação não só de um, mas de diversos valores. Para tanto, as linguagens fornecem uma estrutura também primitiva de criação de variáveis associadas chamadas **vetores** (arrays).

Quando um vetor é alocado, todos os seus valores são dispostos na memória lado a lado (LAFORE, 2005). Por exemplo, ao alocar um vetor de inteiros de 10 posições, o computador irá buscar 40 bytes contínuos



Atividade 2

Desenhe como fica a organização da memória em Java quando se aloca um vetor de 10 números inteiros.

livres na memória (4 bytes por inteiro) e alocá-los em uma única operação. Essa continuidade permite que o acesso sequencial aos dados do vetor seja muito veloz, pois ele maximiza o uso das memórias cache do processador.

Além disso, o acesso por índice a um vetor é praticamente instantâneo. Como os dados estão dispostos lado a lado, as linguagens de programação podem armazenar apenas o primeiro endereço do vetor e descobrir a posição de qualquer outro elemento por uma operação matemática simples. Por exemplo, no caso do mesmo vetor de inteiros, se o seu primeiro elemento estiver na posição 3.328 da memória, como um inteiro tem o tamanho de 4 bytes, saberemos que o elemento de índice 8 estará na posição $3.328 + 8 * 4 = 3.328 + 32 = 3.360$.

Porém, para fins práticos de um software de grande porte, os vetores apresentam uma séria desvantagem: são estáticos, isto é, não mudam de tamanho (LAFORE, 2005). O problema disso é que, na maior parte das vezes, não saberemos quantos dados serão alocados até que comecemos a alocá-los. Por exemplo, se você for cadastrar um grupo de alunos de uma escola no seu sistema, não saberá quantos alunos serão inseridos até que o usuário efetivamente os insira. Mas, sem essa informação, como criar um vetor? A resposta dessa pergunta está neste livro: precisaremos criar um conjunto de estruturas de dados, em que faremos operações como criar, excluir e movimentar vetores e variáveis de maneira eficiente, dando ao programador abstrações mais interessantes, como as listas e os conjuntos.

1.4 Classes



Embora os tipos primitivos sejam úteis, os dados normalmente possuem forte correlação, sendo dificilmente usados de maneira isolada. Por exemplo, em um sistema de biblioteca, os dados do livro, como título, autor e número de páginas, provavelmente aparecerão juntos. Para que isso ocorra, linguagens de programação permitem criar tipos de dados compostos, agrupando dados de tipos diferentes em uma estrutura mais complexa, criada pelo programador. Na linguagem Java, esses tipos de dados compostos são chamados de **classes** (SIERRA; BATES, 2010). Por exemplo:

```
01 public class Livro {  
02     private String nome;  
03     private int paginas;  
04     private String autor;  
05  
06     public Livro(String nome, int paginas, String autor) {  
07         this.nome = nome;  
08         this.paginas = paginas;  
09         this.autor = autor;  
10     }  
11  
12     public String getNome() {  
13         return nome;  
14     }  
15  
16     public int getPaginas() {  
17         return paginas;  
18     }  
19  
20     public String getAutor() {  
21         return autor;  
22     }  
23 }
```

A alocação de memória de um objeto de uma classe é feita por meio do comando new. Quando ele é executado, o compilador calcula o tamanho dos dados de todos os atributos da classe e realiza uma única alocação de toda aquela área no heap. Em seguida, é executado o construtor da classe para inicializar a memória com os valores iniciais determinados pelo programador.

Os dados da classe ficarão armazenados de maneira contínua, lado a lado. Em boa parte das linguagens, a mesma lógica pode ser usada para vetores; porém, esse não é o caso do Java (LAFORE, 2005). Além disso, as classes também possuem um importante papel na organização do código. Neste livro, usaremos as classes extensivamente para organizar as várias estruturas de dados que programaremos.

! Atenção

Por ser muito pequeno, ignoraremos o impacto de performance para fins do nosso estudo.

Por fim, perceba que não falamos em memória ocupada por métodos da classe. Cada objeto de uma classe possui uma única referência para a tabela de métodos virtuais da classe (chamada vtable). Essa tabela tem o endereço de todos os métodos que podem sofrer polimorfismo, ou seja, que, em algum momento, foram sobreescritos nas classes filhas. Isso também implica dizer que acessar um método virtual leva a um pequeno **impacto de performance** (GOODRICH; TAMASSIA, 2013).

1.4.1 Valores e referências

! Atenção

Observe que a referência em si se comporta igual a um tipo primitivo, isto é, funciona por valor. Uma variável local de referência será até mesmo alocada no stack. Mas não a confunda com os dados para onde a referência aponta, ou seja, os dados do objeto, que sempre estarão no heap. Copiar uma referência copia o apontamento, mas não o objeto em si.

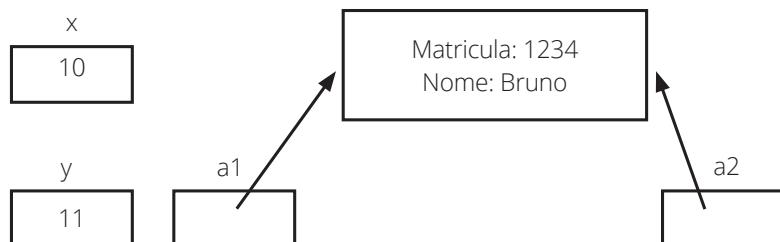
No Java, as variáveis de objetos são chamadas de **referências** (DEITEL, 2010), porque, em vez de guardar todos os dados do objeto, elas contêm apenas um número inteiro com o endereço de memória onde esses dados estão. Quando atribuímos uma referência a outra, esse endereço de memória simplesmente é **copiado** e, portanto, as duas passam a apontar para o mesmo objeto (SIERRA; BATES, 2010). Veja um exemplo:

```
01 var x = 10;
02 var y = x;
03 y = y + 1;
04
05 var a1 = new Aluno(1234, "Vinícius");
06 var a2 = a1;
07 a2.setNome("Bruno");
08 System.out.println(a1.getNome()); //Imprime Bruno
```

A Figura 1, a seguir, demonstra visualmente essas variáveis ao final do código. Observe que, ao invés de um número com o endereço de memória, representamos a referência por meio de uma flecha, apontando para seu conteúdo na memória.

 Figura 1

Valores (x e y) e referências (a1 e a2) na memória



Fonte: Elaborada pelo autor.

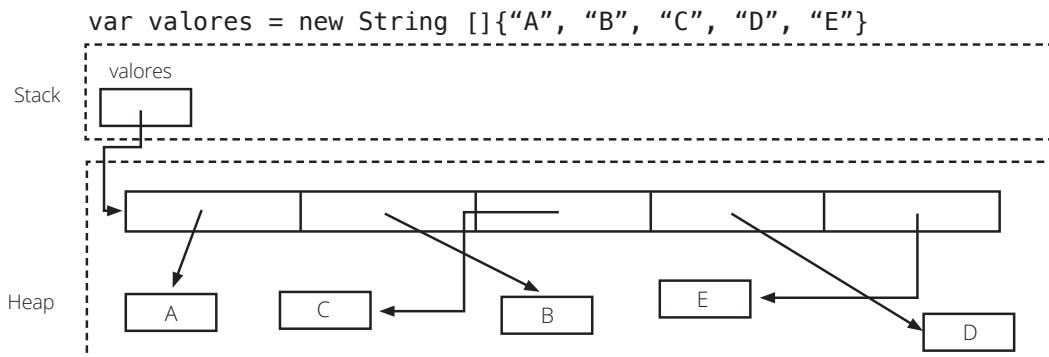
Note que x e y contêm diretamente os valores 10 e 11, pois, como explicado, são variáveis de tipos primitivos e houve uma cópia do valor de x em y antes que a soma fosse feita. Assim, ao final do processo, x e y terminaram com dois valores diferentes. Já as variáveis a1 e a2 somente apontam para a área de memória contendo um objeto da classe Aluno. Portanto, a atribuição não gerou nova alocação e cópia, e o uso de a2 alterou a mesma informação para qual a1 apontava.

Isso tem uma consequência importante: um vetor de objetos em Java será um vetor de referências. Desse modo, os objetos em si ainda estarão espalhados pela memória, pois serão criados em várias alocações diferentes. Assim, os dados não estarão mais lado a lado, o que implica em perda de performance por não aproveitar tão bem o cache. Esse é o pior caso, já que o garbage collector possui algoritmos que tentam alocar os objetos juntos (GOETZ, 2003), mas a garantia de que isso está ocorrendo em sua aplicação, em um dado momento, não existe. Observe um exemplo de alocação de um vetor de objeto (Figura 2).



Figura 2

Alocações de um vetor de objetos



Fonte: Elaborada pelo autor.



Atividade 3

Desenhe como fica a organização da memória em Java quando se aloca uma classe Cliente onde um de seus atributos é um objeto da classe Dependente, com base em:

```
01 public class Cliente {  
02     private int codigo = 123;  
03     private Dependente dependente = new Dependente();  
04 }  
05  
06  
07 public class Dependente {  
08     private int cod = 555;  
09     private int idade = 10;  
10 }
```



Atenção

Perceba que o diagrama simplifica o nome do aluno. Como no Java, o tipo String também é uma classe. O campo Nome conteria uma outra referência e o objeto do nome em si estaria alocado em outra área de memória.

Segundo Main e Savitch (2005), outras linguagens, como a C++ e a C#, possuem instruções específicas para permitir que objetos sejam criados por valor, o que permite criá-los até mesmo como variáveis locais na memória stack.

Finalmente, observe que, embora a variável de referência *valores* tenha sido criada no stack, o vetor em si, para o qual ela aponta, foi criado no heap. Isso porque ele também é considerado um objeto em Java. Por questões de simplicidade, neste livro vamos ainda considerar vetores como estruturas contínuas em memória.



CONSIDERAÇÕES FINAIS

Neste capítulo, vimos que existem diferentes tipos de memória, seja fisicamente ou seja do ponto de vista organizacional. Toda essa variedade exige que programemos estruturas diferentes para acessar essa memória de maneira eficiente.

Vimos também como as variáveis de tipo primitivo organizam-se na memória. Porém, percebemos que essas estruturas são estáticas, isto é, não podem mudar de tamanho uma vez que foram inicializadas. Além disso, essas estruturas não consideram pontos importantes para aplicações reais, como ordenação ou não duplidade de elementos em seu interior.

Fica, desse modo, evidente a necessidade de se estudar bons algoritmos e de viabilizar estruturas inteligentes de organizar dados. Sendo assim, essas estruturas de dados serão o foco dos próximos capítulos.



REFERÊNCIAS

DEITEL, H. M. *Java: como programar*. São Paulo: Pearson, 2010.

GOETZ, B. Garbage collector in the HotSpot JVM: generational and concurrent garbage collection. *IBM*, 25 nov. 2003. Disponível em: <https://www.ibm.com/developerworks/library/j-jtp11253/index.html>. Acesso em: 24 out. 2019.

GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados & algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.

LAFORE, R. *Estruturas de dados & algoritmos em Java*. Rio de Janeiro: Ciência Moderna, 2005.

MAIN, M.; SAVITCH, W. *Data structures & other objects using C++*. 3. ed. Boston: Pearson, 2005.

ORACLE. Primitive Data Types. *The Java Tutorials*, 2019. Disponível em: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>. Acesso em: 24 out. 2019.

SIERRA, K.; BATES, B. *Use a cabeça! Java*. Rio de Janeiro: Alta Books, 2010.

TANENBAUM, A. S.; BOS, H. *Sistemas operacionais modernos*. 4. ed. São Paulo: Pearson Education do Brasil, 2016.



GABARITO

1.

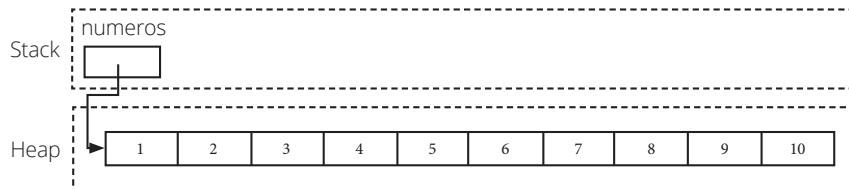
	Organização	Custo (alocação)	Disponibilidade
Stack	Organizada	Baixo	Baixa
Heap	Desorganizada	Alto	Alta

2. Observe que:

- O vetor em si é um objeto. Portanto, será alocado no heap.
- A variável *números* é alocada no stack, mas é uma mera referência ao vetor em si.
- Os valores são tipos primitivos (int). Então, ficam lado a lado na memória e não são referências.

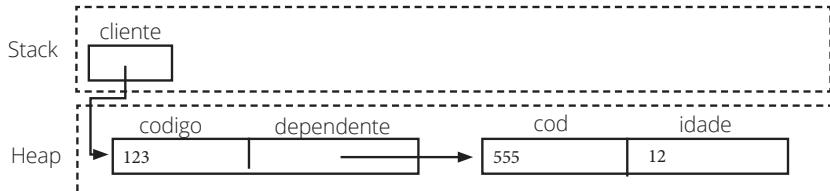
Portanto, o desenho será:

```
var numeros = new int[] {1,2,3,4,5,6,7,8,9,10};
```



3.

```
var cliente = new Cliente();
```



Pilhas e filas

Até agora, vimos os conceitos básicos sobre alocação e sobre a importância das estruturas de dados. Neste capítulo, iniciamos os estudos das estruturas em si. Em todo o livro, vamos organizar o aprendizado da seguinte forma: inicialmente, estudaremos o conceito das estruturas, ou seja, qual é a lógica por trás de seu uso, a sua utilidade etc. Em seguida, veremos as implementações de cada uma delas e suas características técnicas. Essa distinção é importante, pois o mesmo conceito pode ser programado de maneiras diferentes, cada uma com suas próprias características de performance. Damos início a este capítulo por duas estruturas muito básicas: as pilhas e as filas. Então, vamos começar?

2.1 Conceitos



Figura 1

Pilha de camisetas



Quando você passa suas camisetas, provavelmente pega o ferro e coloca-se ao lado de uma bancada vazia. À medida que as camisetas são passadas e dobradas, elas são colocadas nesse espaço vazio, uma sobre a outra, formando uma **pilha de camisetas**.

Quando você move a pilha para seu quarto e vai guardar as camisetas, inicia o processo pelo topo. Assim, a última camiseta inserida na pilha será a primeira a ser retirada.

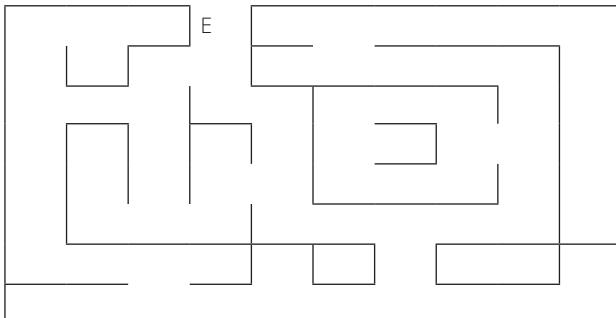
A estrutura de dados *pilha* faz algo similar. Nela, controlamos o acesso aos dados, de modo que as inserções sejam, obrigatoriamente, feitas em sequência, no topo da pilha, e as remoções sejam feitas na ordem inversa das inserções (LAFORE, 2005). A pilha será a primeira estrutura estudada por ser bastante simples.

Um dos exemplos mais clássicos de implementação da pilha é o recurso *voltar*. Ele empilha cada comando dado pelo usuário para que possam ser desempilhados a cada pressionar das teclas CTRL+Z, desfazendo-os.



Estudo de caso

Outro uso interessante da pilha é programar o computador para sair de um labirinto, como o apresentado a seguir.



Para resolver esse problema, o tabuleiro é representado na forma de uma matriz, em que numeramos cada casa.

00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

O algoritmo, então, funciona do seguinte modo: definimos que o computador caminhará em 4 direções diferentes: 0 – esquerda, 1 – cima, 2 – direita, 3 – baixo. Para cada casa caminhada, empilhamos o número da casa e a última direção para a qual o computador foi. Quando o computador se encontrar “preso”, ele desempilha o caminho feito até então, simulando que está “voltando” pelo labirinto.

Na figura ilustrada, o computador começaria fazendo o seguinte caminho:

00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

(Continua)

O computador inicia na casa 03. Como não pode fazer um movimento para a esquerda (movimento 0), passa a tentar o próximo movimento, para cima, e não consegue. Tenta para a direita e, novamente, não consegue. Move, então, para baixo. Assim, empilha 03 (3 – baixo). Agora está na casa 13 e pode mover para a esquerda, empilhando 13 (0 – esquerda). Segue com essa lógica empilhando: 12 (2 – baixo), 22 (0 – esquerda), 21 (0 – esquerda), 20 (1 – cima), 10 (1 – cima), 00 (2 – direita), 01 (2 – direita). Perceba que, assim, ele chega à casa 02, sem saída. Após tentar se deslocar por todas as direções e perceber que não há para onde seguir, ele começará a desempilhar o caminho para retornar de onde veio. Assim, desempilha a casa 01 e “lembra-se” de que a última tentativa nessa casa foi a direita. O que faz, então? Tenta se deslocar para baixo agora, empilha novamente a casa 01 e passa para a casa 11.

00	01	02	03	04	05	06	07	08	09
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Novamente, ele chega a uma encruzilhada. Agora, desempilhará várias casas (11, 01, 00, 10) até chegar à casa 20, onde ainda pode descer. E a lógica segue até que ele saia do labirinto.

Outra estrutura que também fornece acesso a dados de maneira controlada é a **fila**. A diferença desta e da pilha está na ordem em que os elementos são removidos (GOODRICH; TAMASSIA, 2013). Em uma fila, o primeiro elemento inserido será o primeiro a ser removido. Note que entramos em fila o tempo todo no dia a dia, seja em um caixa de supermercado ou no banco. O conceito é exatamente o mesmo.

 Figura 2
Fila de pessoas



Andrey_Popov/Shutterstock

Filas são amplamente usadas na computação, pois, com elas, podemos implementar sistemas de senhas. Também podem ser usadas para enfileirar dados entre duas partes diferentes do sistema. Por exemplo, uma parte do sistema pode receber dados da internet, processá-los e enfileirá-los para que outra parte do sistema utilize esse dado na interface gráfica. Outro exemplo clássico é a fila de impressão, chamada de *spool da impressora*, que enfileira as requisições de impressão feitas a uma única impressora por meio da rede.

Antes de iniciarmos nosso estudo sobre essas estruturas, é preciso saber que podemos classificar as implementações de qualquer estrutura de dados de duas maneiras (LAFORE, 2005). São elas:

- quanto ao **limite de dados**, em que as estruturas podem ser estáticas ou dinâmicas. Estruturas **estáticas** possuem uma quantidade fixa de dados que são capazes de comportar, geralmente definida no momento de sua criação. Estruturas **dinâmicas** não possuem essa limitação, tendo sua capacidade máxima definida apenas pela quantidade de memória disponível para o processo.
- quanto à **disposição dos dados na memória**, em que as estruturas podem ser sequenciais ou encadeadas. Nas estruturas **sequenciais**, os dados são colocados lado a lado na memória, enquanto que, nas estruturas **encadeadas**, os dados se encontram dispersos em vários endereços de memória.

Assim, uma simples estrutura como a pilha poderá conter diversas implementações. Só para se ter uma ideia, a biblioteca padrão da plataforma Java contém seis implementações diferentes de pilha. Neste capítulo, por exemplo, apresentamos as pilhas e filas estáticas, por serem simples. Vamos mostrar, também, a versão encadeada de cada uma delas e comparar as duas abordagens.

2.2 Implementação da pilha



As operações possíveis para as pilhas e filas são:



Operações de pilha e fila

Operação	Descrição
Adicionar	Insere o elemento no topo da pilha/fila.
Remover	Remove o elemento do topo da pilha/fila.

Atenção

Encadear as estruturas é uma das abordagens de implementação para se criar estruturas dinâmicas. Isto quer dizer que uma estrutura encadeada jamais será estática e vice-versa. Por isso, podemos chamar as estruturas simplesmente de **estáticas** ou de **encadeadas** e saberemos que a primeira será necessariamente sequencial e a segunda, necessariamente, dinâmica. Porém, é possível criar estruturas sequenciais dinâmicas.

Saiba mais

Na administração, a operação da pilha é chamada de *UEPS* (último a entrar/, primeiro a sair). Na informática, também encontramos o termo *LIFO* (do inglês, last in, first out). De maneira similar, na fila, essas operações podem ser chamadas de *PEPS* (primeiro a entrar, primeiro a sair) ou *FIFO* (do inglês, first in, first out).

(Continua)

Operação	Descrição
Limpeza	Elimina todos os elementos da pilha/fila.
Verificar se está cheia	Retorna verdadeiro se não há mais espaço para a inserção de elementos na pilha/fila.
Verificar se está vazia	Retorna verdadeiro se não há nenhum elemento na pilha/fila.
Iteração	Consulta elementos da pilha/fila na mesma ordem em que seriam removidos, mas sem removê-los.

Fonte: Elaborado pelo autor.

Em Java, criamos uma determinada interface para essas operações (SIERRA; BATES, 2010). Inicialmente, vamos considerar que os dados inseridos na pilha são números inteiros, mas apresentaremos uma versão mais genérica no final desta seção.

```

1  public interface Pilha {
2      void adicionar(int valor);
3      int remover();
4      boolean isCheia();
5      boolean isVazia();
6
7      void limpar();
}

```

Por enquanto, deixamos a iteração intencionalmente de fora. Embora sua operação seja simples, sua implementação é um pouco mais complexa, por isso, vamos estudá-la no próximo capítulo.

2.2.1 Pilha estática

Vamos criar a classe `PilhaEstatica`, que utiliza o vetor `dados` para armazenar seus elementos. O tamanho do vetor será definido em seu construtor.

Além do vetor, vamos utilizar uma variável inteira chamada `topo`, que conterá o índice do último elemento inserido na pilha. Quando esta estiver vazia, a variável `topo` conterá o valor especial `-1`. Essas definições já permitem implementar o construtor e o método `isVazia`:

```

1  public class PilhaEstatica implements Pilha
2  {
3      private int[] dados;
4      private int topo = -1;
5
6      public PilhaEstatica(int tamanho) {
7          dados = new int[tamanho];
8      }
9
10     @Override
11     public boolean isVazia() {
12         return topo == -1;
13     }

```

Vejamos, a seguir, um exemplo com uma pilha de cinco elementos em três situações distintas: vazia, após a inserção de 3 elementos e cheia.



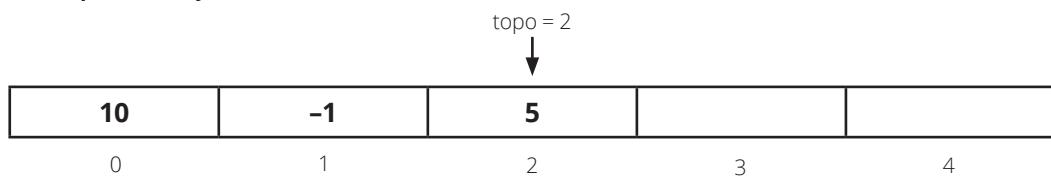
Figura 3

Pilha vazia, pilha com 3 elementos e pilha cheia

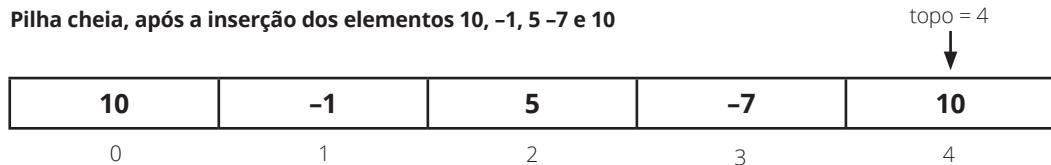
Pilha vazia



Pilha após a inserção dos elementos 10, -1 e 5



Pilha cheia, após a inserção dos elementos 10, -1, 5, -7 e 10



Fonte: Elaborada pelo autor.

Observe que a variável `topo` foi acrescida em 1 a cada elemento inserido, obtendo o valor final de 2 após as três inserções. Podemos inserir elementos até que a pilha esteja cheia, ou seja, até que a variável `topo` contenha o índice do último elemento. No caso de uma

pilha de cinco elementos, o elemento será o índice 4. No caso de uma pilha de n elementos, esse índice será igual ao tamanho do vetor menos um (`dados.length-1`). Portanto, a implementação do método `isCheia` será:

```
1 |     @Override  
2 |     public boolean isCheia() {  
3 |         return topo == dados.length-1;  
4 |     }
```

Atividade 1

A pilha estática pode ter alguns métodos próprios, como `getCapacidade()`, que retorna a quantidade máxima de elementos possível, e `getTamanho()`, que retorna quantos elementos foram inseridos. Faça a implementação desses métodos.

Observe que tentar inserir um elemento em uma pilha já cheia é uma operação inválida, afinal, não haverá espaço no vetor para armazená-lo. Essa tentativa é chamada de **overflow** e, nesse caso, disparamos uma exceção do tipo `IllegalStateException`, indicando que a operação de inserção não é possível quando o estado da pilha for igual a *cheia*. Assim, a versão final do método `adicionar` será:

```
1 |     @Override  
2 |     public void adicionar(int dado) {  
3 |         if (isCheia()) { //Está cheia? Dispara a exceção  
4 |             throw new IllegalStateException("Pilha cheia!");  
5 |         }  
6 |  
7 |         topo = topo + 1; //Movimenta o topo  
8 |         dados[topo] = dado; //Adiciona o elemento no vetor  
9 |     }
```

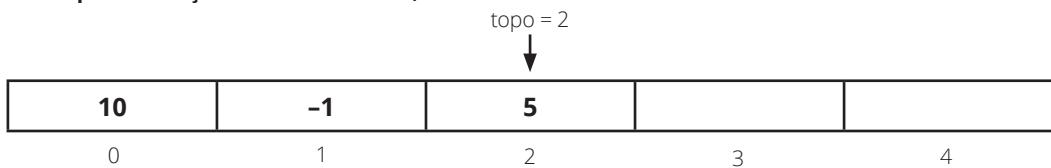
A remoção de um elemento ocorre de maneira muito similar. Observe na figura a seguir.



Figura 4

Pilha após a remoção de um elemento

Pilha após a inserção dos elementos 10, -1 e 5



Pilha após a remoção de 1 elemento



Fonte: Elaborada pelo autor.

Removemos o elemento indicado pela variável topo e, em seguida, reduzimos o seu valor em -1. Similarmente ao que ocorre na inserção, não podemos remover um elemento de uma pilha vazia. Essa operação, também de erro, é chamada de **underflow**. Portanto, a implementação do método remover será:

```
1  @Override
2  public int remover() {
3      if (isVazia()) { //Vazia? Dispara exceção
4          throw new IllegalStateException("Pilha vazia!");
5      }
6
7      int valor = dados[topo]; //Guarda o valor
8      dados[topo] = 0; //Removemos o elemento
9      topo = topo - 1; //Movimenta o topo
10     return valor; //Retorna o valor
11 }
```

Resta-nos apenas uma operação: a de *limpeza da pilha*. No caso da pilha estática, a limpeza pode ser feita simplesmente alterando o valor da variável topo para -1 e removendo as referências dentro do vetor dados.

```
1  @Override
2  public void limpar() {
3      topo = -1;
4      for (int i = 0; i <= topo; i++) {
5          dados[i] = 0;
6      }
7  }
```

Observe que, por um lado, o vetor dados sempre existirá, desde o início da construção da pilha. Assim, ele não deixa de existir quando a pilha é limpa e isso tem uma importante implicação: estruturas estáticas sempre ocupam memória, mesmo que não contenham elementos. Por outro lado, a única estrutura de controle que temos é a variável topo. Esse “valor extra”, que não representa informação de fato dentro da estrutura, é chamado de **overhead**. No caso da pilha estática, o overhead é fixo e muito pequeno (só 4 bytes).

2.2.1.1 Testando a pilha

Vamos criar uma classe para testar nossa pilha? Para isso, basta criar uma nova classe com o método `main`:

```
1 public class Main {  
2     public static void main(String[] args) {  
3         Pilha ps = new PilhaEstatica(10);  
4         //Adiciona os números de 0 a 9  
5         for (int i = 0; i < 10; i++) {  
6             ps.adicionar(i);  
7         }  
8         //Remove os elementos  
9         while (!ps.isVazia()) {  
10            System.out.print(ps.remover() + " ");  
11        }  
12    }  
13 }  
14 }
```



Sugerimos que você teste outras situações, como tentar adicionar mais elementos do que a pilha comporta (overflow) ou remover além do último elemento (underflow).

Esse programa adiciona os números de 0 a 9 na pilha. Em seguida, ele os desempilha até que a estrutura esteja vazia. O resultado será:

```
9 8 7 6 5 4 3 2 1 0
```

2.2.1.2 Generalizando a pilha

Nossa pilha está interessante, mas trabalha apenas com números inteiros. Que tal alterá-la para que suporte qualquer tipo de dado? Fazemos isso por meio do recurso generics, incluído a partir da versão 8 da plataforma Java. Com ele, podemos transformar o tipo de dado que a pilha trabalhará em um parâmetro. Para utilizá-lo, devemos alterar a interface `Pilha` da seguinte forma:

```
1 public interface Pilha<T> {  
2     void adicionar(T valor);  
3     T remover();  
4     boolean isCheia();  
5     boolean isVazia();  
6     void limpar();  
7 }
```

Observe que o tipo de dado `int`, antes fixo, foi substituído pelo tipo `T`. O valor de `T` será substituído por outro tipo de dado posteriormente. Agora, modificamos a classe da pilha de acordo com essa nova interface. A seguir, listamos a classe e os métodos alterados. Os métodos omitidos mantiveram sua implementação inalterada.

```
1 public class PilhaEstatica<T> implements Pilha<T> {  
2     private T[] dados = null;  
3     private int topo = -1;  
4  
5     public PilhaEstatica(int tamanho) {  
6         this.dados = (T[])new Object[tamanho];  
7     }  
8  
9     @Override  
10    public void adicionar(T dado) {  
11        if (isCheia()) { //Está cheia? Dispara a exceção  
12            throw new IllegalStateException("Pilha cheia!");  
13        }  
14  
15        topo = topo + 1; //Movimenta o topo  
16        dados[topo] = dado; //Adiciona o elemento no vetor  
17    }  
18  
19    @Override  
20    public T remover() {  
21        if (isVazia()) { //Vazia? Dispara exceção  
22            throw new IllegalStateException("Pilha vazia!");  
23        }  
24  
25        T valor = dados[topo]; //Guarda o valor  
26        topo = topo - 1; //Movimenta o topo  
27        return valor; //Retorna o valor  
28    }  
}
```

Agora, basta modificar a classe `Main` que utiliza a pilha. A modificação é bastante simples; basta alterar a construção da pilha para:

```
Pilha<Integer> ps = new PilhaEstatica<>(10);
```



Desafio

Experimente criar pilhas de outros tipos; por exemplo, para adicionar textos com o tipo `String`.

2.2.2 Pilha encadeada

Agora que você já entendeu a pilha estática, vamos estudar uma abordagem radicalmente diferente: a pilha encadeada. Embora mais complexa, a pilha encadeada é também dinâmica, removendo totalmente a restrição de tamanho da pilha e limitando a quantidade de elementos apenas devido à memória disponível para o programa (GOODRICH; TAMASSIA, 2013).

Em uma pilha encadeada, cada dado é colocado em uma estrutura conhecida como **nó**, que é uma pequena classe contendo o valor a ser adicionado à pilha e uma referência para o elemento anterior. Assim, cada nó se liga a outro, e basta que a classe da pilha mantenha a referência para o nó do topo.



Fonte: Elaborada pelo autor.

Podemos criar o nó como uma classe interna estática (SIERRA; BATES, 2010) da classe PilhaEncadeada:

```
1 |     private static class No<T> {
2 |         public No anterior;
3 |         public T dado;
4 |     }
```

Observe que a pilha estará vazia quando não houver nós, ou seja, quando a referência do topo tiver o valor `null`. Assim, podemos iniciar as primeiras implementações da classe PilhaEncadeada:

```
1 |     public class PilhaEncadeada<T> implements Pilha<T> {
2 |         private static class No<T> {
3 |             public No anterior;
4 |             public T dado;
5 |         }
6 |
7 |         private No<T> topo = null;
```

(Continua)

```

8
9     @Override
10    public boolean isCheia() {
11        return false;
12    }
13
14    @Override
15    public boolean isVazia() {
16        return topo == null;
17    }

```

Observe que o método `isCheia` sempre retornará falso, já que o limite para a quantidade de dados dessa pilha não será conhecido. De fato, a memória costuma ser tão grande que, exceto nos casos de programas em que há muitos dados sendo processados de uma vez, será considerada infinita para fins práticos. Já o método `isVazia` retornará verdadeiro sempre que o topo não estiver apontando para nenhum nó.

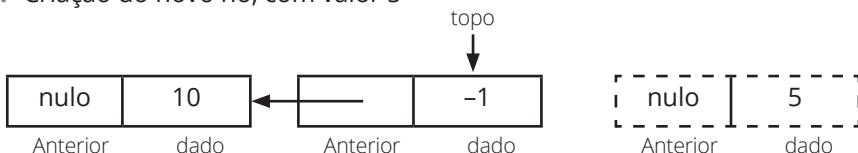
Vamos analisar agora o processo de adição na pilha. Para adicionar um elemento, precisamos criar um novo nó, fazer com que ele considere o topo atual como seu nó anterior e atualizar o topo da pilha, conforme descrito na Figura 6.



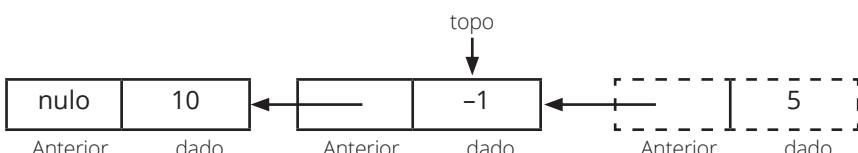
Figura 6

Processo de adição na pilha encadeada

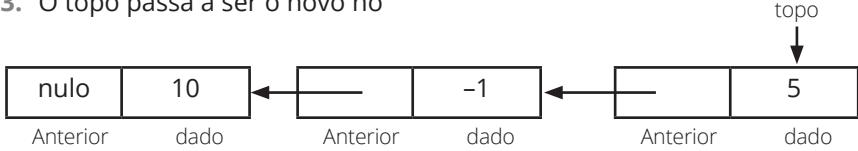
1. Criação do novo nó, com valor 5



2. Anterior do novo nó passa a ser o topo atual



3. O topo passa a ser o novo nó



Fonte: Elaborada pelo autor.

Em código, isso será implementado como:

```
1  @Override
2  public void adicionar(T valor) {
3      var novo = new No<T>(); //1. Cria o no
4      novo.dado = valor; //e associa o dado
5      novo.anterior = topo; //2. Associa o anterior ao topo
6      topo = novo; //3. Atualiza o topo
7  }
```

Atividade 2

Qual seria o custo para implementar um método `getTamanho()` na pilha encadeada? Como essa mesma implementação difere da pilha estática?

Vídeo

Na análise de algoritmos, frequentemente utilizamos a notação assintótica (também conhecida como notação Big-O) para descrever o custo das operações. Esse vídeo, do professor Eduardo Mendes, descreve o funcionamento dessa notação.

FUNDAMENTOS para notação assintótica e contagem de instruções | Análise de Algoritmos. 2018. 1 vídeo (13 min.). Publicado pelo canal Eduardo Mendes. Disponível em: <https://www.youtube.com/watch?v=uxt09pB7nzw>. Acesso em: 25 nov. 2019.

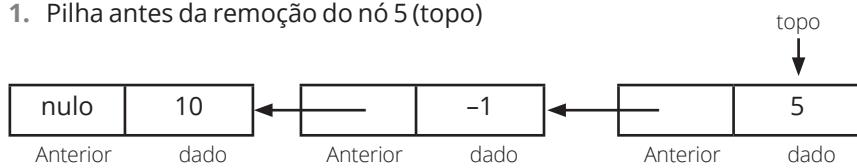
Observe que, como a pilha nunca fica cheia, não há exceção disparada para a situação de overflow. Além disso, note que essa pilha faz uma nova alocação a cada novo elemento, o que implica em um custo maior nessa operação. Contudo, em vez de alocar previamente toda memória para a pilha, como faz a pilha estática, a memória é ocupada gradualmente. Por fim, considere o overhead da pilha encadeada. Para cada dado associado, será necessário gastar 4 bytes adicionais para a referência de seu nó anterior. Além disso, há também os 4 bytes para a referência do topo. Assim, uma pilha com 50 elementos gastará $50 * 4 + 4 = 204$ bytes em dados adicionais de controle. Por último, mais uma coisa que precisamos notar sobre essa pilha é que os dados ficam dispersos pela memória, pois cada alocação ocorrerá em momentos diferentes.

Analisemos agora o processo de remoção da pilha. Para remover um dado, basta retornar o dado do topo e, então, fazer com que a variável `topo` passe a referenciar o seu nó anterior (Figura 7).

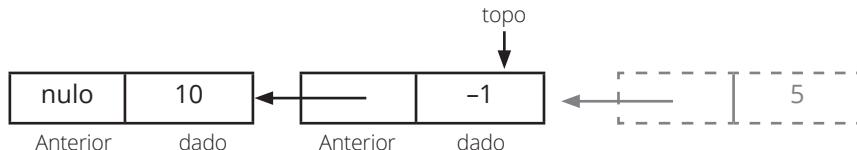
Figura 7

Remoção do elemento 5 na pilha encadeada

1. Pilha antes da remoção do nó 5 (topo)



2. Atualiza o topo para o penúltimo nó



Fonte: Elaborada pelo autor.

Porém, ao contrário do que ocorre na adição, é necessário testar o caso de underflow, ou seja, verificar se não há a tentativa de remoção em uma pilha vazia. Observe a implementação em código.

```
1 |     @Override
2 |     public T remover() {
3 |         if (isVazia()) { //Vazia? Dispara exceção
4 |             throw new IllegalStateException("Fila vazia!");
5 |         }
6 |         T dado = base.dado;
7 |         base = base.proximo;
8 |         if (base == null) {
9 |             topo = null;
10 |         }
11 |         return dado;
12 |     }
```

Observe que, ao executarmos a linha 10 do código, o antigo nó do topo da pilha não terá mais referências, isto é, não haverá nenhuma variável no programa apontando para ele. Isso fará com que ele seja coletado pelo garbage collector e deixe de ocupar memória no computador. Em uma linguagem não gerenciada, como o C++, será necessário desalocar essa memória explicitamente com o comando `delete` (LAFORE, 2005).

Finalmente, podemos implementar a função `limpar`. Observe que o vínculo de todos os nós com a classe pilha é a variável `topo`. Portanto, para limpar os dados, basta que definamos seu valor para `null`. Como não haverá referências para todos os blocos da pilha, o garbage collector eliminará da memória todos os nós. Sendo assim, o código dessa função será simplesmente:

```
1 |     @Override
2 |     public void limpar() {
3 |         topo = null;
4 |     }
```

Observe que, em uma linguagem não gerenciada, seria necessário percorrer cada um dos elementos da pilha, chamando o `delete` sobre os nós. Veja, a seguir, como ficaria o código nessa linguagem (MAIN; SAVITCH, 2005):

```
1 |     while (topo != nullptr) {  
2 |         auto anterior = topo.anterior;  
3 |         delete topo;  
4 |         topo = anterior;  
5 |     }
```

Portanto, apesar de não ser explícito em Java, devemos considerar que a limpeza da pilha possui um custo alto devido à desalocação. Isso é diferente do que ocorre com a pilha estática, na qual esse custo é zero.

2.2.2.1 Testando a pilha

Agora, basta testar o funcionamento da pilha. Observe que, para isso, precisaremos alterar apenas uma única linha em nosso Main, aquela da criação da pilha:

```
Pilha<Integer> ps = new  
PilhaEncadeada<>();
```

O resultado será exatamente o mesmo:

```
9 8 7 6 5 4 3 2 1 0
```

Isso reforça a diferença entre a implementação das classes `PilhaEncadeada` e `PilhaEstatica` e seu conceito, expresso pela interface `Pilha`.

2.3 Implementação da fila



Vamos agora à implementação das filas. Como dito anteriormente, as operações básicas das filas são exatamente as mesmas da pilha. Portanto, a interface **fila** será:

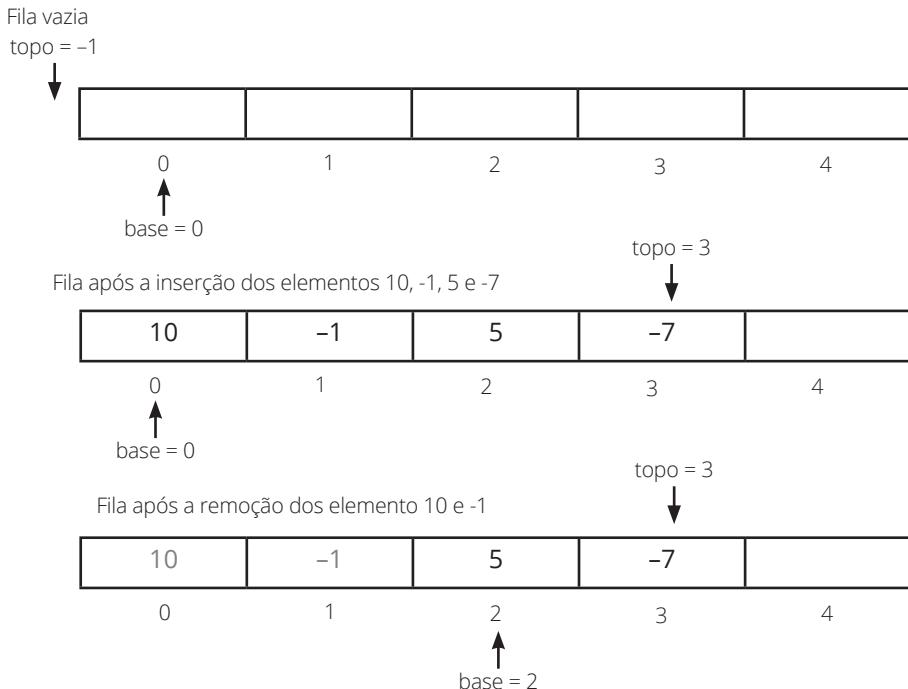


2.3.1 Fila estática

No caso da fila, precisamos controlar dois tipos de elemento: o **topo**, onde o último elemento da fila está, e a **base**, que marca a posição do primeiro elemento. Observe na figura a seguir:

 Figura 8

Operações na fila estática



Fonte: Elaborada pelo autor.

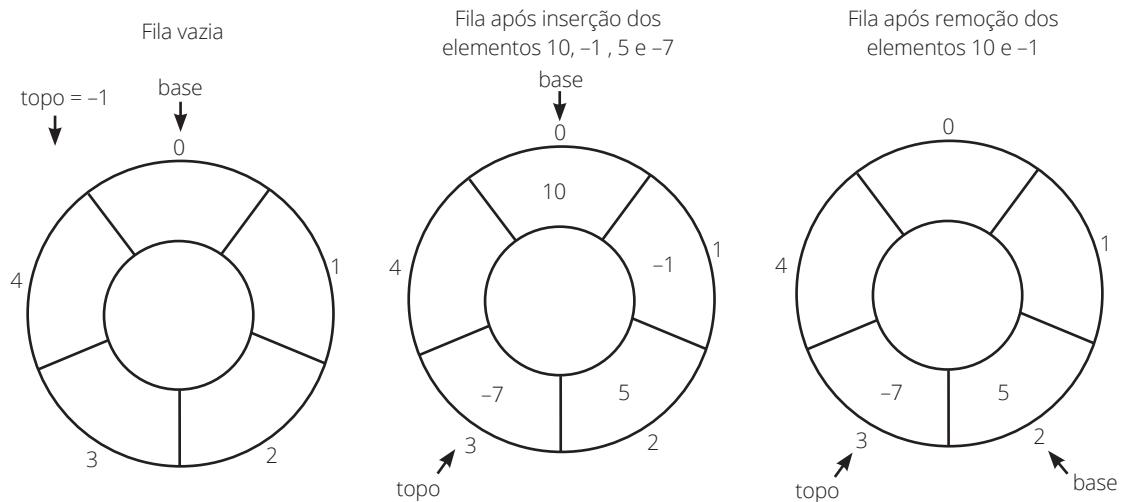
Observe que o topo possui o valor especial -1 para indicar que a fila está vazia. Porém, olhe atentamente as outras linhas da Figura 8. O que ocorreria se tentássemos inserir mais três elementos? Recairíamos em uma situação de overflow, mesmo tendo espaço no início da fila.

Como poderíamos corrigir isso? Uma estratégia seria movimentar todos os elementos para a esquerda (MAIN; SAVITCH, 2005). Porém, pode se tornar uma operação bastante custosa em termos de processamento, pois a fila poderia conter milhares de elementos. Uma alternativa mais inteligente seria utilizar os elementos iniciais, como se o vetor fosse circular (GOODRICH; TAMASSIA, 2013); assim, não é necessário deslocar nenhum elemento. Por isso, a implementação estática da fila é chamada de **fila circular**. Observe essa operação a seguir.



Figura 9

Operações na fila circular



Fonte: Elaborada pelo autor.

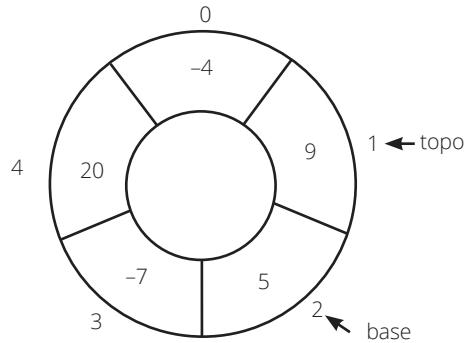
E o que aconteceria se tentássemos adicionar mais três elementos nessa fila? O resultado seria o seguinte (Figura 10):



Figura 10

Adição de mais 3 elementos à fila circular

Fila após a adição dos elementos 20, -4 e 9



Fonte: Elaborada pelo autor.

Observe que, para “ultrapassar o último elemento”, o topo deslocou-se novamente para o início. Vamos, então, implementar o construtor da fila e o método `mover`, que realiza essa operação.

```

1  public class FilaCircular<T> implements Fila<T> {
2      private int base = 0;
3      private int topo = -1;
4
5      private T[] dados;
6
7      public FilaCircular(int tamanho) {
8          this.dados = (T[]) new Object[tamanho];
9      }
10
11     private int mover(int posicao) {
12         return posicao+1 == dados.length ? 0 : posicao+1;
13     }

```

Observe que o método `mover` é privado, uma vez que ele é parte da mecânica interna da fila.

Analisemos, então, as condições de fila vazia e fila cheia:

- A fila estará vazia sempre que o topo for -1. Observe que essa é uma condição especial, e teremos que forçá-la quando o último elemento for removido. Nessa situação, também definiremos a base para 0.
- A fila estará cheia se ela não estiver completamente vazia e o topo estiver posicionado antes da base, conforme podemos observar na Figura 10. A forma mais fácil de testar isso é calcular seu movimento e verificar se o valor resultante coincide com a base.

Portanto, os métodos `isVazia` e `isCheia` são implementados da seguinte maneira:

```

1  @Override
2  public boolean isVazia() {
3      return topo == -1;
4  }
5
6  @Override
7  public boolean isCheia() {
8      return !isVazia() && mover(topo) == base;
9  }

```

Para adicionar um valor, basta movimentarmos o topo e, então, inserirmos o elemento na posição resultante. É importante também testar a situação de overflow.

```
1  @Override
2  public void adicionar(T valor) {
3      if (isCheia()) { //Está cheia? Dispara a exceção
4          throw new IllegalStateException("Fila cheia!");
5      }
6
7      topo = mover(topo);
8      dados[topo] = valor;
9  }
```

Já a remoção será feita movimentando a base e, caso seja o último elemento a ser removido, devemos colocar as variáveis topo e base em seus valores iniciais (o equivalente a limpar a fila). Além disso, também será necessário considerar a condição de underflow.

```
1  @Override
2  public T remover() {
3      if (isVazia()) { //Vazia? Dispara exceção
4          throw new IllegalStateException("Fila vazia!");
5      }
6
7      T dado = dados[base];
8      if (base == topo) { //Último elemento?
9          limpar();
10     } else {
11         dados[base] = null;
12         base = mover(base);
13     }
14     return dado;
15 }
16
17 @Override
18 public void limpar() {
19     base = 0;
20     topo = -1;
21     for (int i = 0; i < dados.length; i++) {
22         dados[i] = null;
23     }
24 }
```

Observe que, de modo similar à pilha estática, essa implementação possui as seguintes características (MAIN; SAVITCH, 2005):

- O overhead é pequeno: apenas dois inteiros para o controle do topo e da base (totalizando 4 bytes).
- A memória é pré-alocada com o tamanho de todos os elementos possíveis. Entretanto, o custo de remoção e adição é praticamente nulo, uma vez que novas alocações/desalocações não são feitas.
- Os elementos se encontram lado a lado na memória.

A fila circular é um exemplo de como uma implementação inteligente pode poupar o uso de memória sem desperdiçar grandes recursos de processamento.

Desafio

Que tal testar agora a impressão na fila por conta própria?
Tente exibir todos os valores enfileirados exatamente como fizemos com a Pilha. Outra sugestão é testar as situações de underflow e overflow para ver o que acontece.

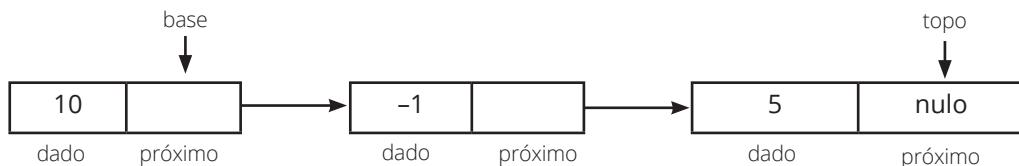
2.3.2 Fila encadeada

Vejamos agora a implementação da fila encadeada. Sua execução é muito similar à da pilha encadeada. Assim como ela, a fila também possuirá um nó, porém, ele apontará para o próximo, e não para o anterior. Além disso, também precisaremos controlar o nó da base, onde ocorrerão as exclusões, e o nó do topo, onde ocorrem as inserções (GOODRICH; TAMASSIA, 2013).



Figura 11

Fila encadeada com 3 elementos



Fonte: Elaborada pelo autor.

A Figura 11 ilustra essa situação. Nela, a variável topo aponta para o último nó, contendo o valor 5 no campo `dado` e com o campo `proxímo` de valor nulo (já que, por ser o topo, não há próximo nó). A variável base aponta para o primeiro nó, com valor 10, e o campo `proxímo` aponta para o nó seguinte.

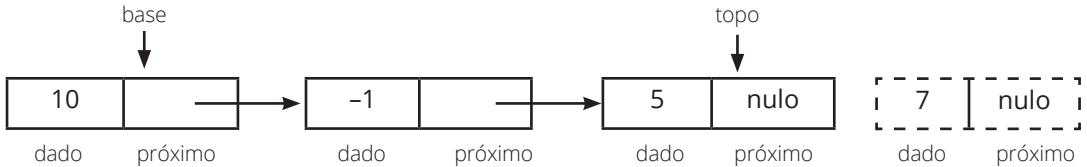
Inicialmente, tanto o topo quanto a base valem nulo, ou seja, não apontam para nenhum nó. Essa condição pode ser usada para testar a situação de underflow. Para adicionarmos um elemento na fila, criamos um novo nó. Em seguida, fazemos o nó do topo apon-

tar para o nó criado e, finalmente, atualizamos o topo. É preciso tomar cuidado no primeiro nó adicionado, já que, nessa situação, também devemos considerá-lo a base. Essas operações são descritas na figura a seguir.

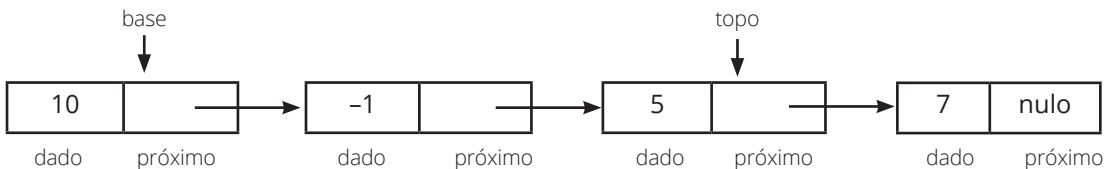
 **Figura 12**

Inserção do valor 7 na fila encadeada

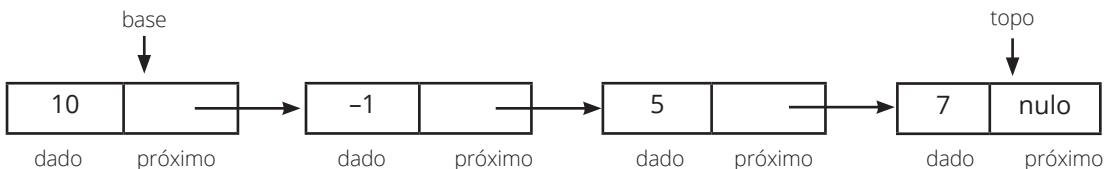
1. Criação do novo nó, com valor 7



2. Nó do topo aponta para o nó criado



3. Atualiza o topo



Fonte: Elaborada pelo autor.

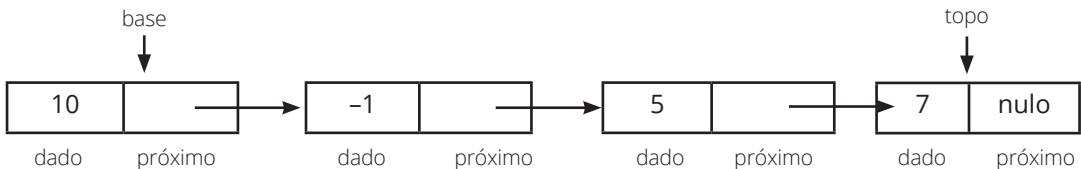
A remoção de um nó é feita retornando o dado atualmente apontado pela base. Em seguida, fazemos com que esta seja atualizada pelo valor marcado em seu campo `proximo`. Caso se torne nula, a fila passa a ficar vazia e, assim, definimos também o valor da variável `topo` para nulo. Na remoção, é necessário, inclusive, testar a situação de underflow. A figura a seguir mostra uma remoção (Figura 13).



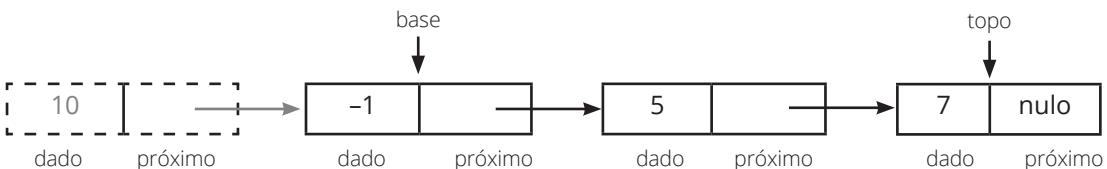
Figura 13

Remoção na fila encadeada

1. Fila antes da remoção do nó com valor 10 (base)



2. Atualiza a base para o próximo nó



Fonte: Elaborada pelo autor.

A operação de limpeza também é bastante simples; basta definir tanto a variável **base** quanto o **topo** para **null**.

Quanto às características de performance da fila, temos uma situação similar à da pilha encadeada (MAIN; SAVITCH, 2005):

- a memória alocada para os nós também se encontra dispersa na memória;
- a memória também é alocada gradualmente a cada elemento inserido e desalocada a cada elemento removido;
- o overhead também é de 4 bytes (tamanho da referência `proximo`) para cada nó mais 8 bytes das variáveis inteiras `topo` e `base`. Assim, uma fila com 50 elementos ocupará $50 * 4 + 8 = 208$ bytes em dados associados.

Sentiu falta do código? Ele é uma das atividades deste capítulo! Assim, você pode conferir a implementação completa da `FilaEncadeada` no gabarito.

www. Site

A biblioteca de coleções do Java possui uma interface para filas, além de algumas implementações-padrão. Confira essa interface na documentação oficial da linguagem, disponível nesta referência. Use o recurso de tradução do seu navegador, se necessário.

ORACLE. *Interface Queue*. Disponível em: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Queue.html>. Acesso em: 25 nov. 2019.



Atividade 3

Implemente a classe `FilaEncadeada`.



CONSIDERAÇÕES FINAIS

Neste capítulo, vimos duas estruturas de dados básicas, a pilha e a fila, assim como duas implementações diferentes para cada uma delas. Obviamente, há outras execuções dessas estruturas, como adicionar o conceito de prioridade à fila, por exemplo, permitindo que elementos mais impor-

tantes “passem à frente”. Outras implementações permitem que pilhas e filas sequenciais também sejam dinâmicas, recriando os vetores e copiando dados sempre que necessário. O importante é reconhecer que existem compromissos de projeto feitos em cada caso e temos de equilibrar o uso de dados, performance e recursos.

REFERÊNCIAS

- GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados & algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.
- LAFORE, R. *Estruturas de dados e algoritmos em Java*. Rio de Janeiro: Ciência Moderna, 2005.
- MAIN, M.; SAVITCH, W. *Data structures & other objects using C++*. 3. ed. Boston, MS: Pearson Education do Brasil, 2005.
- SIERRA, K.; BATES, B. *Use a cabeça, Java!* Rio de Janeiro: Alta Books, 2010.

GABARITO

1. Adicionar esses dois métodos na classe `PilhaEstatica`:

```
1  public int getTamanho() {  
2      return topo;  
3  }  
4  
5  public int getCapacidade() {  
6      return dados.length;  
7  }
```

2. Seria necessário contar todos os elementos da pilha, um a um. Esse custo é muito maior do que simplesmente retornar o valor da variável `topo`, como seria feito na pilha estática.

- 3.

```
1  public class FilaEncadeada<T> implements Fila<T> {  
2      private static class No<T> {  
3          T dado;  
4          No proximo;  
5      }  
6  
7      private No<T> base = null;  
8      private No<T> topo = null;  
9  }
```

(Continua)

```

9
10    @Override
11    public void adicionar(T valor) {
12        var no = new No<T>(); //1
13        no.dado = valor;      //1
14
15        if (isVazia()) {
16            topo = no;
17            base = no;
18        } else {
19            topo.proximo = no; //2
20            topo = no;         //3
21        }
22    }
23
24    @Override
25    public T remover() {
26        if (isVazia()) { //Vazia? Dispara exceção
27            throw new IllegalStateException("Fila vazia!");
28        }
29        T dado = base.dado;
30        base = base.proximo;
31        if (base == null) {
32            topo = null;
33        }
34        return dado;
35    }
36
37    @Override
38    public boolean isCheia() {
39        return false;
40    }
41
42    @Override
43    public boolean isVazia() {
44        return base == null;
45    }
46
47    @Override
48    public void limpar() {
49        base = null;
50        topo = null;
51    }
52}

```

3

Listas

No capítulo anterior, estudamos as estruturas de dados de acesso controlado: as pilhas e as filas. Em ambas, não havia qualquer liberdade sobre a posição onde os elementos eram incluídos, nem de onde retirá-los.

Neste capítulo, vamos estudar uma estrutura mais flexível: a lista. Ela é uma sequência numerada de itens, acessíveis por índice. Todos nós já trabalhamos com uma estrutura simples de lista estática: os vetores. Porém, aplicações reais geralmente exigem listas dinâmicas e algumas capacidades, como escolher a posição dos elementos a serem inseridos ou removidos. Essas são questões que serão abordadas neste capítulo. Veremos também como agrupar as coleções já implementadas, percorrer a lista por meio de iteradores, e fornecer métodos padrão que facilitam seu uso.

3.1

Conceitos



Figura 1

Lista de compras de supermercado

Uma lista é um conjunto de elementos, acessíveis por meio de seus índices (GOODRICH; TAMASSIA, 2013). Assim, ela garante que os elementos respeitem uma ordem de inserção, seja colocando-os ao final, na ordem em que forem inseridos, ou respeitando a posição imposta pelo programador.

Utilizamos listas extensivamente em nosso dia a dia. Um exemplo clássico é a lista de compras.

Quando escrevemos esse tipo de lista, geralmente adicionamos cada item um embaixo do outro. Embora essa seja a operação mais comum, muitas vezes, incluímos um item no meio da lista, “espremendo-o” entre outros dois, quando lembran-

maçã
laranja
leite
pão
ovos
queijo
detergente

SevenFour/Shutterstock

mos de um item esquecido que provavelmente estará próximo a outros no mercado. Por exemplo, podemos lembrar que temos que comprar banana e adicionar o item entre a laranja e o leite, pois já vamos estar na seção de frutas. Também, quando estamos realizando a compra, podemos acessar os elementos da lista em qualquer ordem, sem nos perdermos, já que vamos riscando os produtos da lista à medida que os encontramos nas prateleiras.

Similarmente à pilha e à fila, as listas consideram um conjunto básico de operações:

 **Quadro 1**
Operações da lista

Operação	Descrição
Limpeza	Elimina todos os elementos da lista.
Adicionar	Adiciona um elemento à lista. São fornecidas duas versões geralmente: uma que permite escolher a posição a ser inserida, e outra que sempre adiciona o elemento ao final da lista.
Remover	Remove o elemento que contém um índice específico.
Testar se cheia ou vazia	Verifica se a lista está vazia ou se está cheia.
Alterar (set)	Substitui um elemento em uma posição por outro.
Obter (get)	Obtém um elemento em um determinado índice.
Tamanho	Retorna a quantidade de elementos da lista (getTamanho).
Busca	Encontra o índice de um elemento dentro da lista. Geralmente são fornecidas duas versões de busca: para a primeira ocorrência (indice) e para a última (ultimoIndice).
Iteração	Permite consultar os elementos da lista em ordem sequencial, sem removê-los.

Fonte: Elaborado pelo autor.

A interface da lista será, portanto:

```

1  public interface Lista<T> {
2      //Informações
3      boolean isVazia();
4      boolean isCheia();
5      int getTamanho();
6
7      //Inclusão de itens
8      void adicionar(T valor);
9      void adicionar(int pos, T valor);
10
11     //Exclusão de itens
12     void limpar();
13     T remover(int pos);

```



Atenção

Observe que as operações de limpeza, adição, verificação de cheia/vazia e iteração são idênticas às da pilha e da fila. Essa repetição nos permite agrupar os três conceitos em uma interface comum, ajuste que faremos ao final deste capítulo.

(Continua)

```

14
15      //Acesso direto a valores da lista
16      T get(int pos);
17      void set(int pos, T valor);
18
19      //Métodos para buscar itens
20      int indice(T valor);
21      int ultimoIndice(T valor);
22  }

```

Não confunda o tamanho da lista com a sua capacidade. O **tamanho** refere-se à quantidade de elementos dentro da lista. Já a **capacidade** diz respeito à quantidade máxima de elementos que podem ser adicionados à lista, ou seja, ao tamanho do vetor dados (LAFORE, 2005). Para fins práticos, é o tamanho que interessará ao usuário da lista; afinal, seus índices terão valores no intervalo de 0 até o tamanho da lista menos um.

Assim como fizemos com a fila e a pilha, implementaremos as versões estática, com vetores, e encadeada, com nós. Por fim, note que a iteração foi novamente deixada de fora. Para implementá-la, precisaremos primeiro entender o uso da interface `Iterator` do Java. Mas não se preocupe; incluiremos ela no decorrer deste capítulo!

3.2

Implementação da lista estática



Vamos iniciar estudando a classe `ListaEstatica`. A lista estática é uma estrutura sequencial que utiliza um vetor de tamanho fixo para armazenar os elementos (GOODRICH; TAMASSIA, 2013).

Essa classe conterá o vetor `dados` e a variável `tamanho`, similarmente ao que ocorre com a pilha. A diferença está no fato de que a variável `tamanho` indica a quantidade de elementos inseridos, e não o índice do último elemento, como no caso da variável `topo` da pilha. Assim, teremos o seguinte código inicial:

```

1  public class ListaEstatica<T> implements Lista<T> {
2      private T[] dados;
3      private int tamanho;
4
5      public ListaEstatica(int capacidade) {
6          this.dados = (T[]) new Object[capacidade];
7      }

```

Apesar de não aparecer de maneira explícita, a variável tamanho tem tamanho 0, pois é o valor de inicialização padrão das variáveis inteiros em Java (DEITEL, P.; DEITEL; H., 2009).

3.2.1 Informações da lista

Vamos às condições de lista vazia e cheia. A lista estará vazia sempre que não contiver elementos, ou seja, sempre que o tamanho for igual a 0. Ela estará cheia quando a variável tamanho atingir a capacidade máxima da lista, ou seja, for igual ao tamanho do vetor dados. Assim, os métodos são implementados da seguinte forma:

```
1  @Override
2  public int getTamanho() {
3      return tamanho;
4  }
5
6  public int getCapacidade() {
7      return dados.length;
8  }
9
10 @Override
11 public boolean isVazia() {
12     return getTamanho() == 0;
13 }
14
15 @Override
16 public boolean isCheia() {
17     return getTamanho() == getCapacidade();
18 }
```

Observe que houve a adição do método getCapacidade para retornar a última informação.

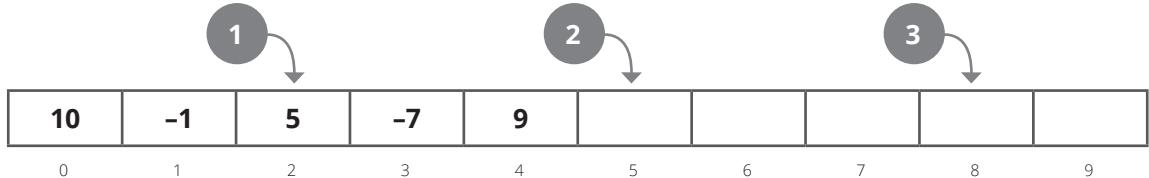
3.2.2 Adição de elementos

Agora, vamos analisar o processo de adição na lista. Em primeiro lugar, é bom lembrarmos que nossa lista armazena objetos; portanto, um dos valores possíveis para uma posição da lista é o valor nulo. Quando adicionamos à lista, permitimos ao programador escolher em qual índice a adição ocorrerá.

Sabemos que índices não podem ser negativos, pois, por definição, os índices da lista começam em zero. Além disso, eles não podem ser superiores à capacidade da lista. Porém, há outros casos, em que os índices estão em uma posição possível dentro do vetor dados. Há três situações, ilustradas pela figura a seguir:

 Figura 2

Lista de tamanho 5 e capacidade 10



1. Qualquer valor entre 0 e o tamanho da lista menos um refere-se a uma posição já ocupada pela lista. Portanto, precisaremos reposicionar os elementos para a direita para acomodar um novo valor.
2. Um valor de índice exatamente igual ao tamanho da lista recairá no fim da lista, após o último elemento.
3. Um índice de valor superior ao tamanho e inferior à capacidade cairá em uma posição vazia, mas, ainda, dentro do vetor **dados**. Nesse caso, precisamos decidir se essa adição é válida. Esse é um exemplo de decisão de projeto. Não há uma abordagem certa ou errada, apenas diferentes abordagens. Permiti-la faria com que a operação **adicionar** ampliasse a lista em um tamanho superior a 1. Além disso, colocaria implicitamente valores nulos entre a posição do último elemento e o elemento inserido. Assim, se um elemento for adicionado à posição 8, as posições 5, 6 e 7 teriam que valer nulo, e o tamanho da lista mudaria de 4 para 8. Por ser um comportamento menos consistente com a ideia de se inserir um elemento só, iremos impedir a operação.

Fonte: Elaborada pelo autor.

Assim, para a inserção em nossa lista, o usuário de nossa classe precisará decidir um índice de valor entre 0 e tamanho. Caso um índice inválido seja informado, dispararemos uma exceção do tipo **IndexOutOfBoundsException**:

```
1 | if (pos < 0 || pos > tamanho) {  
2 |     throw new IndexOutOfBoundsException("Posição inválida!");  
3 | }
```

Na verdade, essa é uma operação tão comum que o Java incluiu uma função que faz a mesma coisa na classe **Objects**, chamada **Objects.checkIndex**. Ela aceita dois parâmetros, o índice e um tamanho (que não é considerado um índice válido). Portanto, o **if** anterior pode ser substituído por:

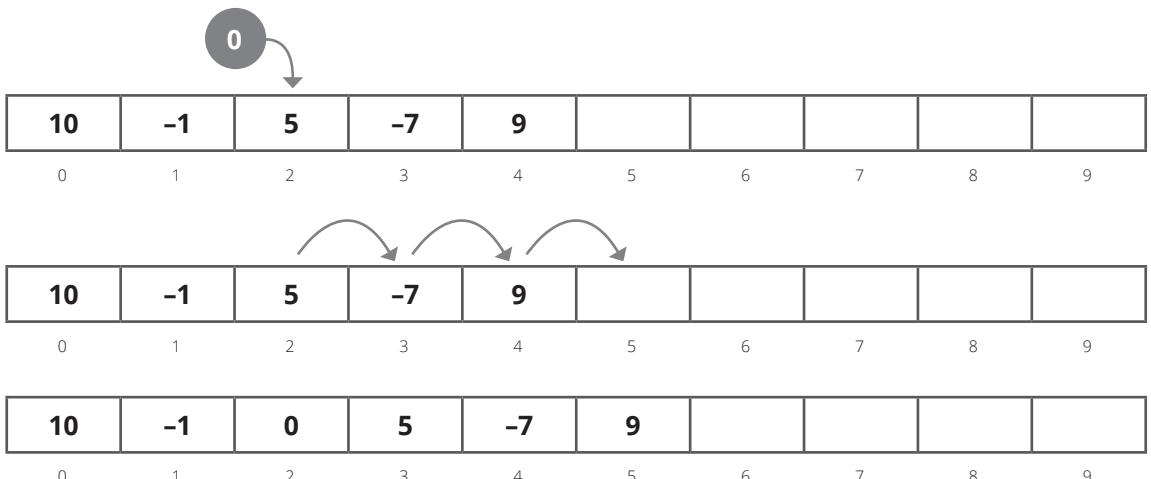
```
Objects.checkIndex(pos, tamanho+1);
```

E como podemos deslocar os elementos da lista? Fazemos isso por meio de um **for**, que percorre o vetor de trás para frente, iniciando no último elemento, copiando-o para a posição seguinte e terminando na posição em que queremos inserir. Depois de colocar o dado, basta adicionar um ao tamanho da lista, conforme ilustrado a seguir:



Figura 3

Inserção do número 0 na posição 2 da lista



Fonte: Elaborada pelo autor.

O código completo da função adicionar é listado a seguir:

```
1  @Override
2  public void adicionar(int pos, T valor) {
3      if (isCheia()) {
4          throw new IllegalStateException("Lista cheia!");
5      }
6      Objects.checkIndex(pos, tamanho+1);
7
8      //Move os dados para a direita
9      for (int i = tamanho-1; i >= pos; i--) {
10         dados[i+1] = dados[i];
11     }
12
13     dados[pos] = valor; //Insere o dado
14     tamanho = tamanho + 1; //Aumenta o tamanho
15 }
```

Além desta, vamos incluir também uma segunda versão do método adicionar, que insere um elemento ao final da lista, sem que seja necessário informar um índice. Essa versão tem um uso mais conveniente, já que, na maior parte das vezes, simplesmente queremos adicionar elementos, sem nos preocuparmos com a posição exata que ele ocupará. Essa versão pode ser facilmente implementada chamando a função que acabamos de criar:

```

1 | @Override
2 | public void adicionar(T valor) {
3 |     adicionar(tamanho, valor);
4 | }

```

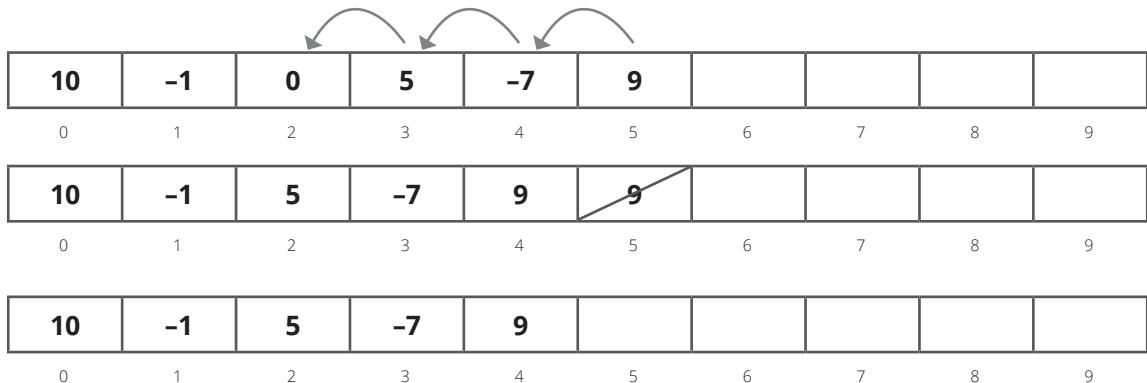
Observe que adicioná-lo ao final da lista tem uma vantagem adicional: o custo de inserção é próximo a 0, já que nenhum elemento será deslocado,

3.2.3 Remoção de elementos

A operação de remoção é bastante similar. Para excluirmos um elemento, teremos que deslocar os elementos posteriores para a esquerda, sobre o elemento excluído. Após isso, definiremos o elemento do final da lista para nulo, para garantir que não haja uma referência a ele fora da lista. A figura a seguir ilustra essa situação:

 Figura 4

Remoção do elemento de índice 2 da lista



Fonte: Elaborada pelo autor.

Podemos traduzir essa lógica no seguinte código:

```

1 | @Override
2 | public T remover(int pos) {
3 |     Objects.checkIndex(pos, tamanho);
4 |
5 |     T dado = dados[pos]; //Dado a ser retornado
6 |     //Move os elementos para a esquerda
7 |     for (int i = pos; i < tamanho-1; i++) {
8 |         dados[i] = dados[i+1];
9 |     }

```

(Continua)

```

10
11     dados[tamanho-1] = null; //Elimina o último dado
12     tamanho = tamanho - 1; //Reduz o tamanho da lista
13     return dado;
14 }
```

Outra operação útil de remoção é a limpeza da lista. De modo similar ao que ocorre com a pilha, basta remover a referência a todos os elementos e zerar a variável `tamanho`:

```

1 @Override
2 public void limpar() {
3     for (int i = 0; i < getTamanho(); i++) {
4         dados[i] = null;
5     }
6     tamanho = 0;
7 }
```

O uso do `for` para preencher o array também é muito comum. Por isso, o Java criou o método `Arrays.fill`, capaz de realizar a mesma tarefa:

```
Arrays.fill(dados, 0, getTamanho(), null);
```

Caso os índices não sejam informados, o método limpará o array inteiro.

3.2.4 Acesso direto aos dados da lista

Obter um dado sem removê-lo (`get`) ou substituir um dado em uma posição específica (`set`) são operações mais simples. Para fazê-las, basta validar o índice informado e, se ele estiver dentro da lista, utilizar o elemento correspondente do vetor:

```

1 @Override
2 public T get(int pos) {
3     Objects.checkIndex(0, tamanho);
4     return dados[pos];
5 }
6
7 @Override
8 public void set(int pos, T valor) {
9     Objects.checkIndex(pos, tamanho);
10    dados[pos] = valor;
11 }
12 }
```

Observe que essas operações são equivalentes ao operador de colchetes dos vetores.

3.2.5 Busca na lista

Uma operação importante é localizar o índice de um elemento dentro da lista. Entretanto, lembre-se de que um mesmo elemento pode estar presente em mais de uma posição. Por isso, essa operação é fornecida em duas versões:

indice

retorna a primeira ocorrência do elemento na lista

ultimoIndice

retorna a última ocorrência do elemento na lista

Ambas as funções retornam o valor negativo `-1` caso o elemento não seja encontrado. Observe que não nos preocupamos em criar funções para retornar os elementos que estejam entre esses dois índices. Isso ocorre pois essas são as duas operações mais comuns e, para casos mais avançados, sempre podemos iterar sobre a lista.

Importante

A busca será feita pelo conceito de *igualdade*, e não *identidade*. No Java, o conceito de *identidade* é testado pelo operador `==`. Dois objetos são idênticos se eles se referirem à mesma instância. Portanto, duas instâncias diferentes contendo o mesmo valor serão consideradas diferentes. Já a igualdade procura por objetos de mesmo valor, mesmo que, na prática, sejam instâncias de objetos diferentes, e é implementada por meio do método `equals`. Isso implica no fato de que o método `equals` deve ser corretamente sobreescrito nas classes utilizadas pelo usuário de nossa classe de lista.

Temos que lembrar que um dos valores possíveis para a lista é o valor nulo, que só pode ser comparado por identidade. Para compararmos duas referências `a` e `b` de maneira segura, podemos fazer o seguinte:

- Testamos pela identidade, ou seja, se `a == b`. Se ambas forem idênticas, obrigatoriamente serão iguais. Se ambas forem nulas, também serão iguais. Isso é uma otimização interessante, já que a comparação com `equals` pode ser custosa e a eliminamos no caso de objetos idênticos.
- Testamos se o objeto `a` é diferente de nulo. Nesse caso, utilizamos o `equals` de `a` em `b`. O próprio método `equals` retornará falso se `b` for nulo. Caso contrário, fará a comparação de valores, retornando verdadeiro se forem iguais. Observe que se `a` for nulo, `b` não será e, portanto, retornará falso.

Todos esses testes podem ser sintetizados em uma única expressão lógica:

```
1 | public static boolean equals(Object a, Object b) {  
2 |     return (a == b) || (a != null && a.equals(b));  
3 | }
```

Essa comparação é tão comum que o Java incluiu essa função por padrão na classe `Objects`. Assim, a busca por um item é feita iterando sobre a lista:

```
1 | @Override  
2 | public int indice(T valor) {  
3 |     for (int i = 0; i < tamanho; i++) {  
4 |         if (Objects.equals(valor, dados[i])) {  
5 |             return i;  
6 |         }  
7 |     }  
8 |  
9 |     return -1;  
10| }  
11|  
12| @Override  
13|  
14| public int ultimoIndice(T valor) {  
15|     for (int i = tamanho-1; i >= 0; i--) {  
16|         if (Objects.equals(valor, dados[i])) {  
17|             return i;  
18|         }  
19|     }  
20|     return -1;  
21| }
```

Assim, finalizamos os métodos básicos da interface da lista. Vamos então à iteração.

3.2.6 Iteração

A operação de iteração envolve percorrer a lista, elemento a elemento. No caso da lista estática, isso pode ser feito por meio de um `while`, como no código a seguir:

```

1 | int atual = -1;
2 |
3 | //Há um próximo elemento?
4 | while ((atual+1) < getTamanho()) {
5 |     //Move para o próximo elemento e o retorna
6 |     atual = atual + 1;
7 |     T dado = dados[atual];
8 |
9 |     //Usa o elemento
10 |    System.out.println(dado);
11 |

```

 Saiba mais

Observe que, se estivéssemos fazendo uma iteração sobre a fila encadeada (do capítulo anterior), faríamos um while com operações similares, mesmo que implementadas de um jeito diferente:

```

1 | No<T> atual = null;
2 |
3 | //Há um próximo elemento?
4 | while (!isVazia() && atual != topo) {
5 |     //Move para o próximo elemento e o retorna
6 |     atual = (atual == null ? base : atual.proximo);
7 |     T dado = atual.dado;
8 |
9 |     //Usa o elemento
10 |    System.out.println(dado);
11 |

```

As operações presentes em qualquer iteração são:

- 1 Posicionar-se antes do primeiro elemento.
- 2 Testar se há um próximo elemento e, caso haja:
 - a Saltar para o próximo elemento.
 - b Retorná-lo.
- 3 Repetir o passo 2.

Isso é importante, pois nos permite encapsular qualquer iteração em uma classe responsável por isso. Assim, independentemente da es-

trutura interna da coleção (encadeada, sequencial etc.), permitiremos que a iteração seja feita pelo seu usuário de maneira homogênea. Esse é o princípio do padrão de projetos Iterator (GAMMA *et al.*, 2007).

Para permitirmos que nossas coleções sejam iteráveis, iniciamos implementando a interface Iterable. Essa interface exige apenas a criação de um método, que retorna um objeto do tipo Iterator (SIERRA; BATES, 2010):

```
Iterator<T> iterator();
```

Por sua vez, o objeto iterador deve implementar dois métodos:

hasNext

Diz se há um dado ainda na coleção sobre a qual ele está iterando.

next

Salta para o próximo elemento e o retorna.

Opcionalmente, também podemos implementar o método remove, que remove o último elemento retornado. Por padrão, esse método retorna uma UnsupportedOperationException. A implementação é interessante em diversas estruturas, pois a iteração pode ser o melhor momento para a remoção de um elemento.

Vamos implementar o iterador em nossa lista. Vamos tornar a interface Lista uma extensão de Iterable:

```
public interface Lista<T> extends Iterable<T> {
```

Agora, vamos incluir uma classe interna na lista para fazer a implementação do Iterator:

```
1  private class ListaIterator implements Iterator<T> {
2      private int atual = -1;
3
4      @Override
5      public boolean hasNext() {
6          return (atual+1) < getTamanho();
7      }
```

(Continua)

```

4  @Override
5  public boolean hasNext() {
6      return (atual+1) < getTamanho();
7  }
8
9  @Override
10 public T next() {
11     atual = atual + 1;
12     return dados[atual];
13 }
14 }
```

Compare essa implementação com o while exibido no início dessa seção. Notou as similaridades?

O último passo é incluir o método iterator na ListaEstatica:

```

1  @Override
2  public Iterator<T> iterator() {
3      return new ListaIterator();
4  }
```

Observe que, agora, aquele while poderia ser reescrito da seguinte forma:

```

1  var iterator = iterator();
2  while (iterator.hasNext()) {
3      T dado = iterator.next();
4      System.out.println(dado);
5  }
```

Perceba que, se você implementar o iterador para a Pilha e para a Fila, o while será exatamente igual para essas estruturas. Isso permitiu que os projetistas da plataforma Java criassem o comando for each, capaz de utilizar o iterador. Assim, a lista agora pode ser percorrida da seguinte forma:

```

1  for (var valor : lista) {
2      System.out.println(valor);
3  }
```

Esse for pode ser lido como “para cada valor no iterável lista”. Além disso, a interface Iterator contém o método padrão forEach, que chama uma função a cada elemento da lista. Assim, o for também

pode ser implementado em uma única linha através do uso de lambda (ORACLE, 2019b):

```
lista.forEach(valor -> System.out.println(valor));
```

Por fim, a interface `Iterator` também diz que o método `next` deve disparar uma `NoSuchElementException`, mesmo que não haja um próximo elemento a ser percorrido. Omitimos esse detalhe de nossas implementações para deixar os códigos mais simples, mas você pode adicionar o seguinte `if` na primeira linha do método se quiser prever esse caso:

```
if (!hasNext()) throw new NoSuchElementException();
```

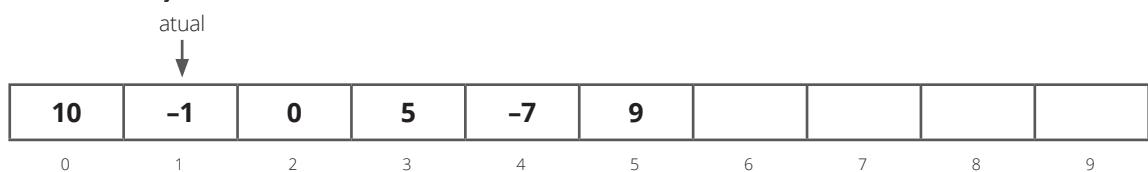
E como poderíamos implementar o método `remove` do `Iterator`? No caso dessa lista, ele pode ser facilmente implementado chamando o método `remover` da classe da lista. Porém, se fizermos isso, teremos que também atualizar a variável `atual`, já que a remoção força o deslocamento dos elementos da lista para a esquerda. Essa situação é mostrada na figura a seguir:



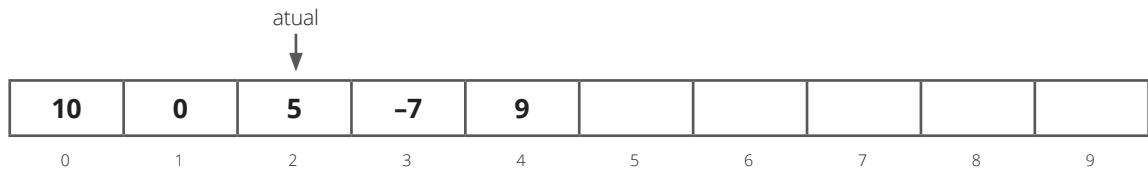
Figura 5

Remoção do elemento de posição 1 da lista estática

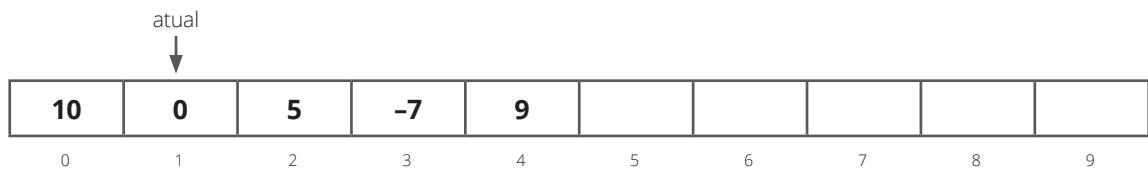
Antes da remoção



Após chamar remover (atual-1) para remover o último retornado



Posição do atual corrigida



Fonte: Elaborada pelo autor.



Atividade 1

Faça a implementação dos iteradores das pilhas e das filas criadas no Capítulo 2. Observe que, no caso dessas duas estruturas, não há remoção, pois a ordem da iteração respeita a ordem de remoção das estruturas.

Assim, a implementação do método `remove` do `Iterator` é feita da seguinte forma:

```
1  @Override
2  public void remove() {
3      remover(atual);
4      atual = atual - 1;
5  }
```

3.2.7 Tornando a lista dinâmica

Desafio

Por meio do conceito de *iteração* sobre a lista, crie na interface `List` o método padrão `todosIndices`, que retorna uma lista contendo todos os índices onde um determinado elemento for encontrado.

Até agora, todas as estruturas sequenciais que criamos eram também estáticas. Isso pode dar a falsa ideia de que toda estrutura sequencial é assim. Mas isso não é verdade. Embora toda estrutura estática seja sequencial, o inverso não é necessariamente verdadeiro.

Vamos criar uma nova classe dinâmica chamada `ListaVetor`. Para isso, primeiro devemos copiar a classe `ListaEstatica` e renomear a cópia para `ListaVetor`.

Então, vamos alterar o método `adicionar`, da `ListaVetor`, para copiar os dados para um novo vetor, sempre que a lista atingir sua capacidade máxima. A nova capacidade será 50% maior do que a anterior. A forma mais fácil de realizar cópias de vetores em Java é por meio do método `Arrays.copyOf`, o qual recebe um vetor a ser copiado e seu novo tamanho como parâmetro. O código do método `adicionar` ficará, portanto:

```
1  @Override
2  public void adicionar(int pos, T valor) {
3      //Já está em capacidade máxima?
4      if (getTamanho() == getCapacidade()) {
5          //Redimensiona
6          int novaCapacidade = (int)(getCapacidade() * 1.5);
7          dados = Arrays.copyOf(dados, novaCapacidade);
8      }
9
10     Objects.checkIndex(pos, tamanho+1);
11
12     //Move os dados para a direita
13     for (int i = tamanho-1; i >= pos; i--) {
14         dados[i+1] = dados[i];
15     }
}
```

(Continua)

```

16
17     dados[pos] = valor; //Insere o dado
18     tamanho = tamanho + 1; //Aumenta o tamanho
19 }

```

Observe que o teste `getTamanho() == getCapacidade()`, antes feito pelo método `isCheia`, foi feito diretamente. Isso porque o método `isCheia` agora deve ser alterado para retornar sempre falso. Para o usuário da `ListaVetor`, a lista nunca estará cheia.

Também podemos criar um segundo construtor para essa lista, que cria a `ListaVetor` com a capacidade padrão de dez elementos:

```

1 public ListaVetor() {
2     this(10);
3 }

```

Você pode estar se questionando: quando a lista diminui? É possível implementar a redução da lista alterando o método `remover` e o método `limpar`. Porém, não faremos isso, já que a classe `ArrayList` do Java também não o faz (LAFORE, 2005). No lugar, criaremos o método `ajustar`, que faz com que a lista fique com o tamanho idêntico à sua capacidade:

```
dados = Arrays.copyOf(dados, getTamanho());
```

Essa escolha é uma decisão comum de projeto, implementada não só pelo Java, mas também pelo C++ (MAIN; SAVITCH, 2005). Isso porque, enquanto a operação de adição ocorre frequentemente aos poucos, a exclusão geralmente ocorre em grandes lotes. Assim, a redução da lista pode ser aplicada somente ao final do processo, evitando custos de *performance*.



Desafio

Torne as classes `PilhaEstatica` e `FilaEstatica` dinâmicas utilizando o método descrito neste capítulo.



Saiba mais

Você sabia que é possível estudar as implementações das classes no próprio Java? Para isso, faça o seguinte: Primeiro, abra o IntelliJ Idea e crie um objeto do tipo `ArrayList`. Depois, clique com o botão direito sobre o nome `ArrayList` e acesse os menus `Go to` → `Implementation`. Por fim, você poderá comparar a implementação presente na plataforma com a que fizemos.

3.3 Implementação da lista encadeada

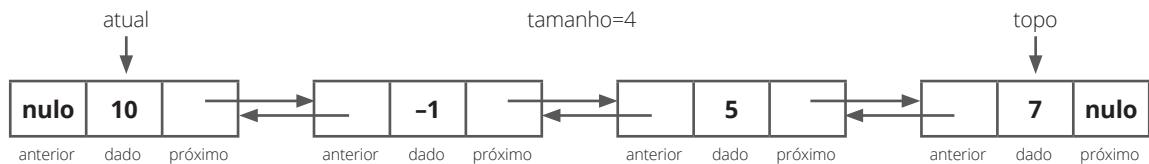


Vamos agora verificar a implementação da lista usando a estratégia duplamente encadeada, isto é, com cada nó contendo uma referência tanto para o próximo nó quanto para o nó anterior. Encadear duplamente permitirá que iteremos sobre a lista nos dois sentidos. Assim, essa classe torna-se muito útil, pois ela pode ser usada como uma pilha ou fila quando necessário. Finalmente, também vamos explorar a estratégia duplamente encadeada porque é essa a estratégia usada na classe `LinkedList`, da biblioteca padrão do Java.

Além disso, também controlaremos o tamanho da lista com a variável inteira `tamanho`, conforme ilustrado na figura a seguir:

 Figura 6

Lista duplamente encadeada com 4 elementos



Fonte: Elaborada pelo autor.

 Saiba mais

A implementação de uma lista simplesmente encadeada também é possível, mas essa classe é muito similar à classe da fila.

Dessa forma, a implementação básica dessa lista será:

```
1  public class ListaEncadeada<T> implements Lista<T> {  
2      private static class No<T> {  
3          No<T> anterior;  
4          T dado;  
5          No<T> proximo;  
6  
7          public No(T valor) {  
8              dado = valor;  
9          }  
10     }  
11  
12     private No<T> base = null;  
13     private No<T> topo = null;  
14     private int tamanho = 0;
```

Note que um construtor para a classe no também foi adicionado.

 Artigo

https://medium.com/@govinda_raj/arraylist-vs-linkedlist-f8c5099153b5

Esse artigo *ArrayList vs LinkedList*, do autor Govinda Raj, publicado no site Medium, em 22 jan. de 2018, compara as duas implementações (ArrayList e LinkedList) em termos de performance, com vários comentários interessantes sobre o assunto. O artigo está em inglês, mas você pode usar o recurso de tradução automática do seu navegador, se necessário.

Acesso em: 2 dez. 2019.

3.3.1 Informações da lista

As informações básicas da lista são implementadas de maneira direta. Afinal, a lista nunca estará cheia por ser dinâmica. Seu tamanho é indicado diretamente pela variável `tamanho`, e ela estará vazia quando o tamanho for 0. Isso nos leva à seguinte implementação:

```
1  @Override
2  public boolean isVazia() {
3      return tamanho == 0;
4  }
5
6  @Override
7  public boolean isCheia() {
8      return false;
9  }
10
11 @Override
12 public int getTamanho() {
13     return tamanho;
14 }
```

Outra possibilidade para testar se a lista está vazia é verificar se a base é nula (`base == null`).

3.3.2 Adição de elementos

Vamos iniciar com a adição ao final da lista. Caso a lista esteja vazia, a operação é simples: as variáveis `topo` e `base` passarão a conter o nó recém-adicionado, e o tamanho da lista se tornará 1. Caso contrário, o nó adicionado será o novo topo da lista, e o atual topo será colocado anteriormente a ele. Isso envolve as seguintes operações:

- 1 A variável `anterior` do nó é atribuída a `topo`;
- 2 A variável `proximo` do nó `topo` é atribuída ao nó adicionado;
- 3 A variável `topo` é substituída pelo nó adicionado e o tamanho adicionado em um.

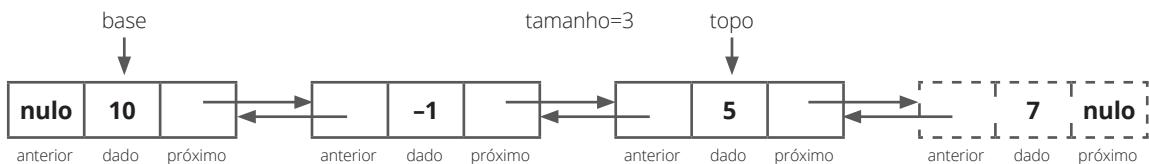
Essa situação é ilustrada na figura a seguir:



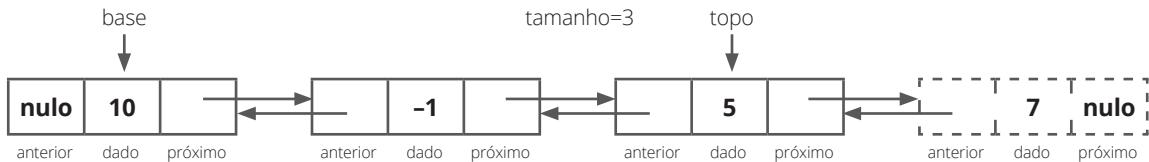
Figura 7

Inserção do valor 7 ao final da lista

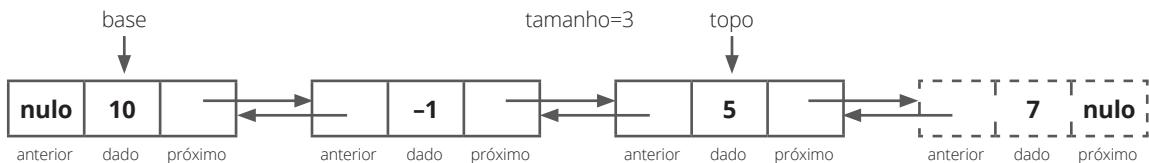
Adicionado o valor 7



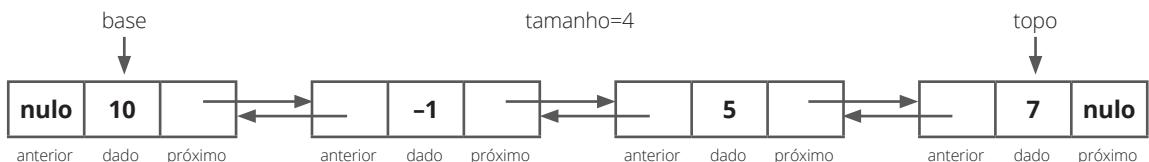
1. no.anterior = topo



2. topo.proximo = no



3. topo = no e atualização do tamano



Fonte: Elaborada pelo autor.

Essa adição pode ser traduzida no seguinte código-fonte:

```

1  @Override
2  public void adicionar(T valor) {
3      var no = new No<T>(valor);
4
5      if (isVazia()) {
6          base = no;
7      } else {
8          no.anterior = topo; //1
9          topo.proximo = no; //2
10     }
11
12     topo = no; //3
13     tamano = tamano + 1; //3
14 }
```

Observe que, diferentemente do que acontece com uma lista sequencial, não é possível acessar diretamente um nó específico. Portanto, para adicionar em um índice, teremos que navegar até o nó em questão, e isso só pode ser feito iterando nó a nó.

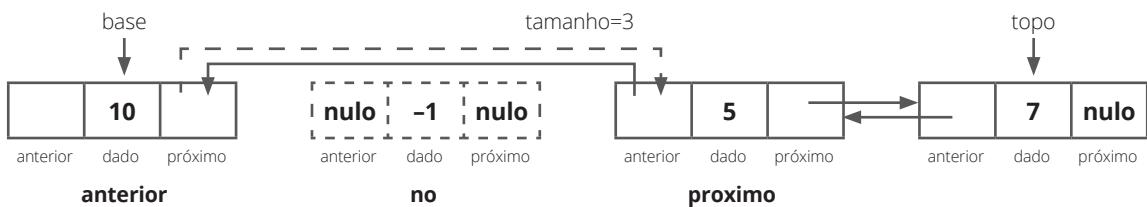
Porém, temos uma otimização possível: se a posição desejada estiver do meio para esquerda da lista, iteramos no sentido natural. Porém, se estiver do meio para direita, iremos obtê-la mais rapidamente, iterando do fim até a posição desejada. Isso garante que, no pior caso, somente saltaremos por meia lista. Vamos implementar o método privado `getNo`, que realiza essa lógica:

```
1 | private No<T> getNo(int pos) {
2 |     Objects.checkIndex(pos, tamanho);
3 |     int meio = tamanho / 2;
4 |
5 |     //Itera para frente
6 |     if (pos <= meio) {
7 |         No<T> atual = base;
8 |         for (int i = 0; i < pos; i++) {
9 |             atual = atual.proxima;
10 |         }
11 |         return atual;
12 |     }
13 |
14 |     //Itera para trás
15 |     No<T> atual = topo;
16 |     for (int i = tamanho-1; i != pos; i--) {
17 |         atual = atual.anterior;
18 |     }
19 |     return atual;
20 | }
```

Observe que sempre lidaremos com três nós, conforme ilustrado na seguinte figura:

**Figura 8**

Inserção do valor -1 na posição 1 da lista



- **no**: o nó que está sendo inserido.
- **proxímo**: o nó que se tornará o próximo ao nó inserido, isto é, o nó que atualmente ocupa a posição de inserção informada pelo usuário.
- **anterior**: o nó que se tornará o anterior ao nó inserido. Antes da inserção, ele se refere ao nó indicado pela variável **anterior** do **no** informado pelo usuário (**proxímo.anterior**).

Fonte: Elaborada pelo autor.

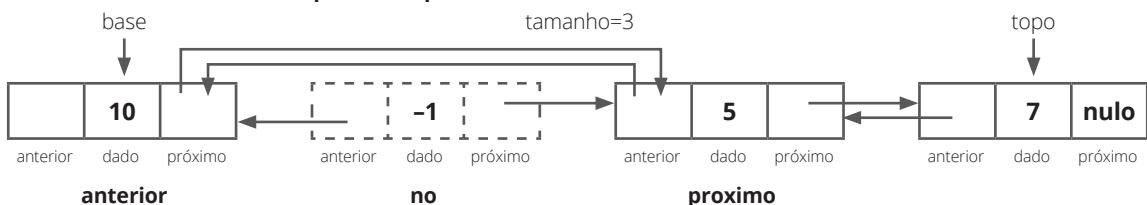
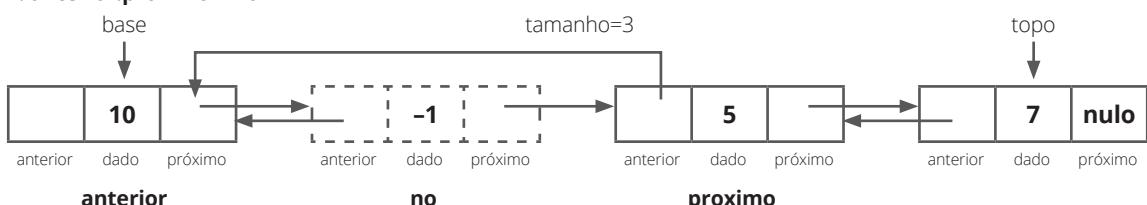
Assim, o processo de inserção é formado pelos seguintes passos:

- 1 Atualizamos os valores de **proxímo** e **anterior** em **no** para referenciarem as variáveis descritas.
- 2 Caso o nó anterior seja nulo, trata-se de uma inserção na base. Nesse caso, atualizamos a variável **base** para apontar para o nó. Caso contrário, atualizamos a referência **anterior.proximo** para o nó sendo inserido.
- 3 Finalmente, atualizamos a variável **proxímo.anterior** para o nó sendo inserido e o tamanho da lista.

Essas operações são ilustradas na figura a seguir:

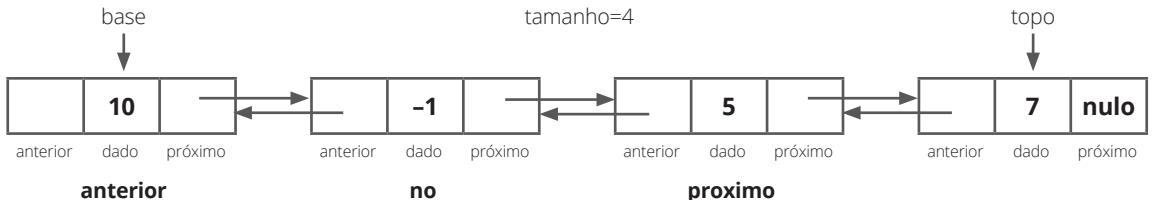
**Figura 9**

Passos para inserção no meio da lista encadeada

1. no.anterior = anterior e no.proximo = proximo**2. anterior.proximo = no**

(Continua)

3. proximo.anterior = no e atualização do tamanho



Fonte: Elaborada pelo autor.

Isso resulta no seguinte código-fonte:

```
1 @Override
2 public void adicionar(int pos, T valor) {
3     Objects.checkIndex(pos, getTamanho()+1);
4
5     //Adicionando no fim?
6     if (pos == getTamanho()) {
7         adicionar(valor);
8         return;
9     }
10
11     var no = new No<T>(valor);
12     var proximo = getNo(pos);
13     var anterior = proximo.anterior;
14
15     no.anterior = anterior; //1
16     no.proximo = proximo; //1
17
18     if (anterior == null) { //Base?
19         base = no;
20     } else {
21         anterior.proximo = no; //2
22     }
23
24     proximo.anterior = no; //3
25     tamanho = tamanho+1; //3
26 }
```

Observe que a inserção, tanto no topo quanto na base dessa lista, tem custo próximo de 0, já que basta a atualização de duas referências no nó que está sendo inserido, uma referência no topo ou base e a atualização no tamanho. Nas demais posições, haverá também o custo de iterar até o elemento que, no pior caso, equivalerá a percorrer metade dos elementos da lista.



Atenção

Para que as situações de topo e lista vazia também fossem incluídas por esse método, testamos, na linha 6, se a inserção é feita ao final da lista e, em caso positivo, encaminhamos o código para a outra versão do método `adicionar`.

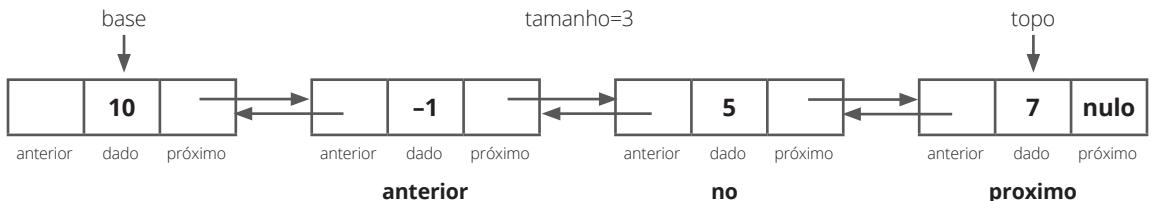
3.3.3 Remoção de elementos

O processo de remoção de elementos ocorre de maneira parecida com o de adição. Assim, localizamos o item a ser removido e atualizamos os campos próximo e anterior dos nós ao seu redor para que ele seja removido. As operações de remoção são ilustradas na figura a seguir:

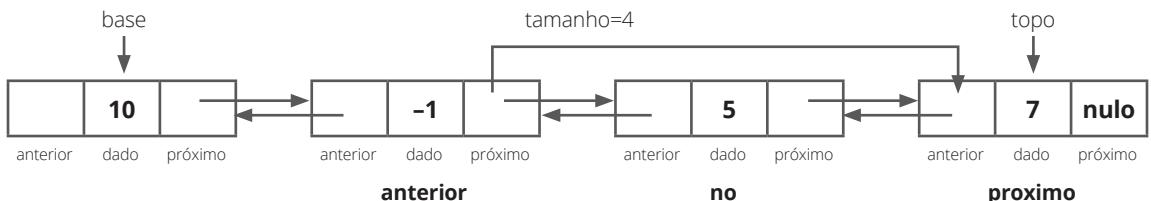
 Figura 10

Exclusão de um nó da lista

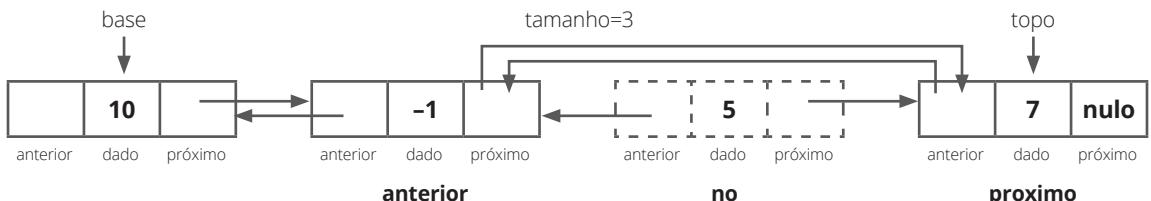
Antes da exclusão do valor 5 da posição 2 da lista



1. anterior.proximo = proximo



2. proximo.anterior = anterior e atualização do tamanho



Fonte: Elaborada pelo autor.

Observe na figura que, no passo 1, há a possibilidade de o nó ser a base da lista e, então, anterior ter valor nulo. Nesse caso, apenas atualizamos a referência base. Analogamente, no passo 2, há a possibilidade da variável proximo ser nula, indicando que a lista está em seu topo e, nesse caso, apenas atualizaremos a variável topo.

```

1 private T remover(No<T> no) {
2     T dado = no.dado;
3     var anterior = no.anterior;
4     var proximo = no.proximo;
5
6     if (anterior == null) {
7         base = proximo;
8     } else {
9         anterior.proximo = proximo; //1
10    }
11
12    if (proximo == null) {
13        topo = anterior;
14    } else {
15        proximo.anterior = anterior; //2
16    }
17
18    tamanho = tamanho - 1; //2
19    return dado;
20 }
21
22 @Override
23 public T remover(int pos) {
24     return remover(getNo(pos));
25 }

```

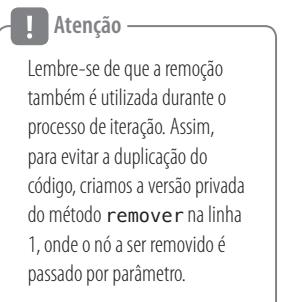
O processo de limpeza da lista é bastante simples. As referências **topo** e **base** são as únicas referências a todos os nós da lista. Por isso, para eliminá-las, basta defini-las para **null**:

```

1 @Override
2 public void limpar() {
3     base = null;
4     topo = null;
5     tamanho = 0;
6 }

```

Lembre-se de que, após esse processo, o garbage collector atuará sobre todos os nós, o que implica no custo de desalocação de memória de toda a lista.



3.3.4 Acesso direto aos dados da lista

Uma das grandes desvantagens da estrutura de nós é que ela impossibilita o acesso direto a um elemento (GOODRICH; TAMASSIA, 2013). Para acessar um índice, seremos sempre obrigados a saltar nó a nó até o elemento desejado. Isso implica em um custo de performance alto, se comparado ao da lista sequencial.

Os métodos `get` e `set` podem ser implementados de maneira simples utilizando a função `getNo`:

```
1  @Override
2  public T get(int pos) {
3      return getNo(pos).dado;
4  }
5
6  @Override
7  public void set(int pos, T valor) {
8      getNo(pos).dado = valor;
9  }
```

Importante

Essa característica torna os iteradores e, consequentemente, o comando `for each` especialmente interessantes. Considere o custo de performance de percorrer a lista encadeada por índices, utilizando um `for` desse tipo:

```
for (int i = 0; i < li.getTamanho(); i++) {
    System.out.println(li.get(i));
}
```

Cada laço de `for` disparará toda a iteração interna do método `get`, o que fará com que toda a lista seja percorrida novamente. Para se ter uma ideia do quanto péssima é essa operação, saiba que a quantidade de elementos acessados seria igual a metade do tamanho da lista **ao quadrado**. Assim, em uma lista com 150 elementos, seriam feitos 11.250 acessos aos nós! Já o iterador saltará sobre cada nó apenas uma vez, fazendo o mesmo trabalho com apenas 150 acessos.

3.3.5 Busca na lista

O processo de busca na lista é muito similar ao que fizemos na função `getNo`. Desse modo, na função `indice`, partimos da base, saltando nó a nó, por meio do campo `proximo`:

```

1  @Override
2  public int indice(T valor) {
3      if (isVazia()) return -1;
4
5      var atual = base;
6      int pos = 0;
7      while (atual != null) {
8          if (Objects.equals(atual.dado, valor)) {
9              return pos;
10         }
11         pos = pos + 1;
12         atual = atual.proximo;
13     }
14
15     return -1;
16 }
```

Observe que essa função também poderia ser implementada fazendo uso do iterador, utilizando a instrução `for each`.

3.3.6 Iteração

Por fim, vamos para a implementação do iterador da lista encadeada. Devemos lembrar que ele deve iniciar antes do primeiro nó. Para isso, iremos criar uma referência ao nó no iterador, chamada `atual` e iniciada em `null`.

Para a condição de próximo nó (`hasNext`), temos as seguintes situações:

- 1 Se a lista estiver vazia, nunca haverá um próximo nó.
- 2 Caso contrário, não haverá um próximo elemento se o atual estiver no topo da lista.

A iteração em si é feita da seguinte forma: se a variável `atual` for nula, o próximo nó será o valor da variável `base`; se a variável `atual` não for nula, o próximo nó é indicado pelo campo `proximo`.

Por fim, para a remoção do nó `atual`, basta chamar o método `remover` e voltar o iterador ao nó anterior, similarmente ao que fizemos com a lista estática. O código completo do iterador é:



Desafio

Que tal implementar a função `ultimoIndice`, partindo do topo (em que `pos=tamanho-1`) e saltando nó a nó por meio do campo `anterior`?



Site

Além da interface Iterator, a plataforma Java também contém a interface ListIterator, que permite a adição de elementos e a iteração reversa. Neste link, você pode consultar a documentação oficial da linguagem Java para conhecê-la melhor! Use o tradutor do seu navegador, se necessário.

Disponível em: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Queue.html>. Acesso em: 2 dez. 2019.

```
1 @Override
2 public Iterator<T> iterator() {
3     return new ListaIterator();
4 }
5
6 private class ListaIterator implements Iterator<T> {
7     private No<T> atual = null;
8
9     @Override
10    public boolean hasNext() {
11        return !isVazia() && atual != topo;
12    }
13
14    @Override
15    public T next() {
16        atual = (atual == null ? base : atual.proxima);
17        return atual.dado;
18    }
19
20    @Override
21    public void remove() {
22        remover(atual);
23        atual = atual.anterior;
24    }
25 }
```

3.4 Últimos ajustes



Você deve ter notado que há muitas similaridades entre as classes implementadas até agora, como:

- podem ser limpas;
- permitem uma forma de adição sem parâmetros;
- possuem os métodos `isVazia` e `isCheia`;
- podem ser iteradas.

Isto ocorre porque essas são características de todas as classes implementadas até agora (pilha, fila e lista), independentemente do seu tipo. Assim, podemos associar todas elas em um único conceito chama-

do **coleção**, isto é, uma classe capaz de armazenar vários objetos. Na programação, mapeamos esse conceito na seguinte interface:

```
1 public interface Colecao<T> extends Iterable<T> {  
2     void limpar();  
3     void adicionar(T valor);  
4  
5     boolean isVazia();  
6     boolean isCheia();  
7 }
```

Podemos então fazer as interfaces *pilha*, *fila* e *lista* herdarem essa interface, removendo delas métodos duplicados. Como exemplo, veja a nova interface da pilha:

```
1 public interface Pilha<T> extends Colecao<T> {  
2     T remover();  
3 }
```

A primeira vantagem de se fazer isso é que o Java permite a inclusão de **métodos padrão** em suas interfaces. Podemos inclui-los para todas as ações que não dependem das partes privadas das coleções, o que geralmente ocorre em métodos que criamos para conveniência de uso, e não para implementar uma funcionalidade em si (ORACLE, 2019a).

Por exemplo, podemos adicionar à interface *Colecao* o método *adicionarTodos*, que recebe como parâmetro um vetor de elementos e o adiciona à coleção:

```
1 default void adicionarTodos(T ... valores) {  
2     for (T valor : valores) {  
3         adicionar(valor);  
4     }  
5 }
```



Atividade 2

Adicione os seguintes métodos padrão na interface *Colecao*:

- **adicionarTodos** similar ao descrito no capítulo, mas que recebe uma coleção qualquer como parâmetro.
- **removerTodos** que remove todas as ocorrências dos elementos recebidos por parâmetro. Igualmente ao que foi feito ao **adicionarTodos**, crie uma versão que recebe por parâmetro um vetor e outra que recebe uma coleção qualquer. Esse método dispara uma **UnsupportedOperationException** no caso das filas e pilhas.
- **contém** que retorna verdadeiro se o **Object** passado por parâmetro estiver dentro da coleção, ou falso, se não estiver.

A segunda vantagem pode ser observada nos próprios métodos padrão: por meio da interface `Colecao`, é fácil permitir que as implementações interoperem. Note que o método `adicionarTodos` pode receber como parâmetro qualquer coleção, seja ela uma Pilha, Fila ou Lista.

Por fim, também é possível agrupar as duas listas sequenciais (`ListaEstatica` e `ListaVetor`) em uma classe abstrata, chamada `ListaSequencial`, deixando seus métodos diferentes (`adicionar` e `isCheia`) abstratos nessa classe. Isso permitirá remover boa parte da duplicidade do código das duas classes.

Observe que há uma separação entre as interfaces `Colecao`, `Pilha`, `Fila` e `Lista`, que representam os conceitos, e as classes concretas (como `ListaEstatica`), que representam as implementações. É considerada uma boa prática utilizar as interfaces em retornos e parâmetros de função na parte pública de nossas classes (BLOCH, 2019), pois isso nos permite alterar a implementação das coleções livremente, sem afetar o resto do sistema.

CONSIDERAÇÕES FINAIS

É difícil imaginar um sistema comercial sem algum tipo de lista. Não é à toa que as classes `ArrayList` e `LinkedList` são parte da biblioteca de coleções do Java desde suas primeiras versões. Afinal, elas são a forma mais comum de se implementar a relação de composição entre objetos. Também não foi por acaso que, neste capítulo, criamos estruturas similares (`ListaVetor` e `ListaEncadeada`). Ao conhecer o funcionamento interno dessas estruturas, temos ciência de que tipo de compromisso de tempo de execução e memória estamos realizando em cada caso.

Também vimos que as listas podem ser usadas de base para a implementação de várias outras coleções. Por exemplo, seria muito fácil implementar uma pilha ou fila utilizando os métodos presentes na `ListaVetor`.

Assim como ocorreu com as pilhas e filas, as implementações apresentadas não são as únicas opções possíveis de listas. A biblioteca Java fornece outras para situações específicas, como a classe `CopyOnWriteArrayList`, adequada para o contexto de paralelismo, além de formas de construir listas sincronizadas e imodificáveis.

No próximo capítulo, veremos maneiras eficientes de tratar elementos ordenáveis, ou seja, elementos cujas grandezas podem ser comparadas entre si.



Atividade 3

Crie a classe abstrata `ListaSequencial`, como descrito no capítulo. As classes `ListaEstatica` e `ListaVetor` devem tornar-se filha dela, eliminando todo código duplicado. Lembre-se de que os métodos `isCheia` e `adicionar` serão abstratos na `ListaSequencial`, portanto, sua implementação será mantida nas classes filhas.



REFERÊNCIAS

- BLOCH, J. *Java Efetivo*: as melhores práticas para a plataforma Java. 3. ed. Rio de Janeiro: Alta Books, 2019.
- DEITEL, P.; DEITEL, H. *Java*: como programar. 8. ed. São Paulo: Pearson do Brasil, 2009.
- GAMMA, E. et al. *Padrões de Projeto*: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2007.
- GOODRICH, M. T.; TAMASSIA, R. *Estruturas de dados & algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.
- LAFORE, R. *Estruturas de dados & algoritmos em Java*. Rio de Janeiro: Ciência Moderna, 2005.
- MAIN, M.; SAVITCH, W. *Data Structures & Other Objects Using C++*. 3. ed. Boston: Pearson Education do Brasil, 2005.
- ORACLE. Default Methods. *The Java Tutorials*, 2019a. Disponível em: <https://docs.oracle.com/javase/tutorial/java/landl/defaultmethods.html>. Acesso em: 19 nov. 2019.
- ORACLE. Lambda Expressions. *The Java Tutorials*, 2019b. Disponível em: <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>. Acesso em: 2 dez. 2019.
- SIERRA, K.; BATES, B. *Use a cabeça, Java!* Rio de Janeiro: Alta Books, 2010.



GABARITO

1. A seguir, listamos os iteradores implementados. Além deles, em cada classe, é necessário implementar o método `Iterator`, retornando uma instância do respectivo iterador, da mesma forma que fizemos para a lista:

```
1 private class PilhaEstaticaIterator implements Iterator<T> {  
2     private int atual = topo+1;  
3  
4     @Override  
5     public boolean hasNext() {  
6         return atual > 0;  
7     }  
8  
9     @Override  
10    public T next() {  
11        atual = atual - 1;  
12        return dados[atual];  
13    }  
14 }
```

```
1 | private class PilhaEncadeadaIterator implements Iterator<T> {
2 |     private No<T> atual = null;
3 |
4 |     @Override
5 |     public boolean hasNext() {
6 |         return !isVazia() && (atual == null || atual.anterior != null);
7 |     }
8 |     @Override
9 |     public T next() {
10 |         atual = (atual == null ? topo : atual.anterior);
11 |         return atual.dado;
12 |     }
13 |
14 }
```

```
1 | private class FilaCircularIterator implements Iterator<T> {
2 |     private int atual = -100;
3 |
4 |     @Override
5 |     public boolean hasNext() {
6 |         return !isVazia() && atual != topo;
7 |     }
8 |
9 |     @Override
10 |     public T next() {
11 |         atual = (atual == -100 ? base : mover(atual));
12 |         return dados[atual];
13 |     }
14 }
```

```
1 | private class FilaEncadeadaIterator implements Iterator<T> {
2 |     private No<T> atual = null;
3 |
4 |     @Override
5 |     public boolean hasNext() {
6 |         return !isVazia() && atual != topo;
7 |     }
8 |
9 |     @Override
10 |     public T next() {
11 |         atual = (atual == null ? base : atual.proximo);
12 |         return atual.dado;
13 |     }
14 }
```

2. Veja a implementação da interface Colecao com os métodos padrão implementados.

Adicionamos um tratamento especial nos métodos adicionarTodos e removerTodos para caso o usuário passe a própria coleção como parâmetro, por exemplo: a.adicionarTodos(a):

```
1  public interface Colecao<T> extends Iterable<T> {
2      void limpar();
3      void adicionar(T valor);
4
5      boolean isVazia();
6      boolean isCheia();
7
8      default boolean contem(Object valor) {
9          var it = iterator();
10         while (it.hasNext()) {
11             if (Objects.equals(it.next(), valor)) {
12                 return true;
13             }
14         }
15         return false;
16     }
17     default void adicionarTodos(T ... valores) {
18         for (T valor : valores) {
19             adicionar(valor);
20         }
21     }
22
23     default void adicionarTodos(T ... valores) {
24         for (T valor : valores) {
25             adicionar(valor);
26         }
27     }
28
29     default void adicionarTodos(Collection<? extends T> valores) {
30         if (valores == this) {
31             throw new UnsupportedOperationException();
32         }
33     }
34 }
```

(Continua)

```
32
33     for (T valor : valores) {
34         adicionar(valor);
35     }
36 }
37
38 default void removerTodos(Colecao<? extends T> valores) {
39     if (valores == this) {
40         limpar();
41         return;
42     }
43
44
45     var it = iterator();
46     while (it.hasNext()) {
47         T valor = it.next();
48         if (valores.contem(valor)) {
49             it.remove();
50         }
51     }
52 }
53
54
55 default void removerTodos(T ... valores) {
56     var lista = new ListaEstatica<T>(valores.length);
57     lista.adicionarTodos(valores);
58     removerTodos(lista);
59 }
60 }
```

3.

```
1 public abstract class ListaSequencial<T> implements Lista<T> {
2     protected T[] dados;
3     protected int tamanho;
4
5     public ListaSequencial(int capacidade) {
6         this.dados = (T[]) new Object[capacidade];
7     }
8
9     @Override
10    public void adicionar(T valor) {
11        adicionar(tamanho, valor);
12    }
13
14    @Override
15    public void set(int pos, T valor) {
16        Objects.checkIndex(pos, tamanho);
17        dados[pos] = valor;
18    }
19
20    @Override
21    public int indice(T valor) {
22        for (int i = 0; i < tamanho; i++) {
23            if (Objects.equals(valor, dados[i])) {
24                return i;
25            }
26        }
27
28        return -1;
29    }
30
31    @Override
32    public int ultimoIndice(T valor) {
33        for (int i = tamanho-1; i >= 0; i--) {
34            if (Objects.equals(valor, dados[i])) {
35                return i;
36            }
37        }
38    }
```

```
39
40     return -1;
41 }
42
43 @Override
44 public T get(int pos) {
45     Objects.checkIndex(0, tamanho);
46     return dados[pos];
47 }
48
49 @Override
50 public void limpar() {
51     Arrays.fill(dados, 0, getTamanho(), null);
52     tamanho = 0;
53 }
54
55 @Override
56 public T remover(int pos) {
57     Objects.checkIndex(pos, tamanho);
58
59     T dado = dados[pos]; //Dado a ser retornado
60
61     //Move os elementos para a esquerda
62     for (int i = pos; i < tamanho-1; i++) {
63         dados[i] = dados[i+1];
64     }
65
66     dados[tamanho-1] = null; //Elimina o último dado
67     tamanho = tamanho - 1; //Reduz o tamanho da lista
68     return dado;
69 }
70
71 @Override
72 public int getTamanho() {
73     return tamanho;
74 }
75
76
```

```
75  public int getCapacidade() {
76      return dados.length;
77  }
78
79  @Override
80  public boolean isVazia() {
81      return getTamanho() == 0;
82  }
83
84  @Override
85  public Iterator<T> iterator() {
86      return new ListaIterator();
87  }
88
89  public void bubbleSort(Comparator<? super T> comparator) {
90      for (int i = 0; i < getTamanho()-1; i++) {
91          for (int j = i+1; j < getTamanho(); j++) {
92              if (comparator.compare(dados[i], dados[j]) > 0) {
93                  T temp = dados[i];
94                  dados[i] = dados[j];
95                  dados[j] = temp;
96              }
97          }
98      }
99  }
100 protected class ListaIterator implements Iterator<T> {
101     private int atual = -1;
102
103     @Override
104     public boolean hasNext() {
105         return (atual+1) < getTamanho();
106     }
107
108     @Override
109     public T next() {
110         if (!hasNext()) throw new NoSuchElementException();
111
112
113
114 }
```

```

115     atual = atual + 1;
116     return dados[atual];
117 }
118
119 @Override
120 public void remove() {
121     remover(atual);
122     atual = atual - 1;
123 }
124 }
125 }

1 public class ListaEstatica<T> extends ListaSequencial<T> {
2     public ListaEstatica(int capacidade) {
3         super(capacidade);
4     }
5
6     @Override
7     public void adicionar(int pos, T valor) {
8         if (isCheia()) {
9             throw new IllegalStateException("Lista cheia!");
10        }
11        Objects.checkIndex(pos, tamanho+1);
12
13        //Move os dados para a direita
14        for (int i = tamanho-1; i >= pos; i--) {
15            dados[i+1] = dados[i];
16        }
17
18        dados[pos] = valor; //Insere o dado
19        tamanho = tamanho + 1; //Aumenta o tamanho
20    }
21
22    @Override
23    public boolean isCheia() {
24        return getTamanho() == getCapacidade();
25    }
26}

```

```
1 public class ListaVetor<T> extends ListaSequencial<T> {
2     public ListaVetor(int capacidade) {
3         super(capacidade);
4     }
5
6     public ListaVetor() {
7         super(10);
8     }
9
10    @Override
11    public void adicionar(int pos, T valor) {
12        //Já está em capacidade máxima?
13        if (getTamanho() == getCapacidade()) {
14            //Redimensiona
15            int novaCapacidade = (int)(getCapacidade() * 1.5);
16            dados = Arrays.copyOf(dados, novaCapacidade);
17        }
18
19        Objects.checkIndex(pos, tamanho+1);
20
21        //Move os dados para a direita
22        for (int i = tamanho-1; i >= pos; i--) {
23            dados[i+1] = dados[i];
24        }
25
26        dados[pos] = valor; //Insere o dado
27        tamanho = tamanho + 1; //Aumenta o tamanho
28    }
29
30    @Override
31    public boolean isCheia() {
32        return false;
33    }
34}
35
```

Ordenação de dados

As coleções vistas até o momento utilizam a ordem de inserção como seu principal critério de ordenação. Esse critério é muito forte para a pilha e a fila, e mais flexível para as listas, sendo, de maneira geral, o que define a ordem em que os elementos serão posicionados.

Entretanto, quando lidamos com sistemas complexos, podemos querer utilizar uma ordenação mais relacionada ao dado em si. Por exemplo, podemos querer ordenar os e-mails de uma caixa de entrada em ordem reversa de data, ou organizar todos os funcionários de uma empresa por ordem de matrícula.

Por isso, o estudo de algoritmos eficientes de ordenação, a criação de estruturas naturalmente ordenadas e a escolha da forma de aproveitar ao máximo essa característica em outras operações (como a busca) se tornam centrais no estudo de estruturas de dados.

Desse modo, a partir deste capítulo, você começará a explorar o conceito de *ordenação*. Vamos iniciar nosso estudo aprendendo a ordenar listas; em seguida, vamos conhecer abordagens para mantê-las sempre ordenadas.

4.1

Conceitos



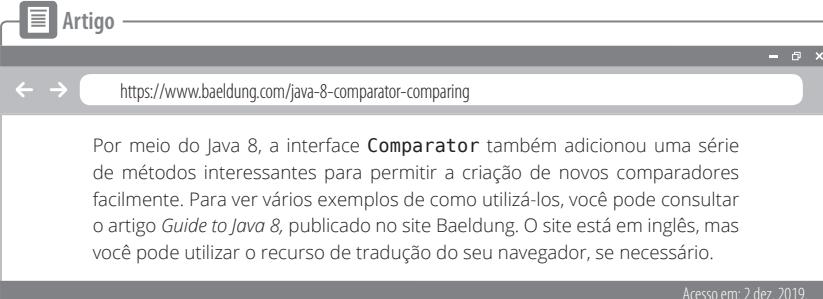
Alguns tipos de dados são comparáveis entre si, isto é, podemos olhar para dois elementos de mesmo tipo e determinar se um deles é maior, menor ou igual a outro (DEITEL, P.; DEITEL, H., 2009). Como essa comparação é possível, podemos colocar esses dados em algum tipo de ordem, seja ela crescente ou decrescente.

Esse conceito é facilmente entendido com alguns dados simples naturalmente ordenados, como números ou datas. Porém, ele pode ser mais complexo quando lidamos com dados compostos, como um aluno ou um carro, pois dados desse tipo podem ser ordenados de várias formas.

Na programação, podemos resolver esse problema por meio de duas estratégias. A primeira é permitir a implementação de uma interface chamada Comparable às classes que possuem uma ordenação natural. E a segunda é fornecer, para as funções de ordenação, um objeto comparador que implementa a interface Comparator, para que elas organizem os dados de acordo com o critério que ele implementa. (DEITEL, P.; DEITEL, H., 2009).

Ambas as interfaces fornecem uma **função de comparação**, isto é, uma função que, dado dois objetos o1 e o2, retorne:

- um número negativo, caso o1 seja menor que o2;
- zero, caso o1 seja igual a o2;
- um número positivo, caso o1 seja maior que o2.



Por meio do Java 8, a interface **Comparator** também adicionou uma série de métodos interessantes para permitir a criação de novos comparadores facilmente. Para ver vários exemplos de como utilizá-los, você pode consultar o artigo *Guide to Java 8*, publicado no site Baeldung. O site está em inglês, mas você pode utilizar o recurso de tradução do seu navegador, se necessário.

Acesso em: 2 dez. 2019.

Além disso, para que seja consistente, essa função deve garantir três propriedades básicas (ORACLE, 2019):

- **Simetria:** o sinal de o1 comparado a o2 deve ser o inverso do sinal de o2 comparado a o1. Isso também implica que a comparação de o1 com o2 deve lançar uma exceção se, e somente se, a comparação de o2 com o1 também lançar exceção.
- **Transitividade:** se o1 comparado a o2 for positivo, e o2 comparado a o3 for positivo, então o1 comparado a o3 também deve ser positivo.
- **Consistência:** se a comparação de o1 com o2 for igual a zero, então a comparação de o1 com o3 deve ser igual à de o2 com o3 para qualquer o3.

Curiosidade

Alguns dados compostos possuem uma ordem natural, como as placas de carro. Porém, isso é uma exceção; na maioria dos casos, eles possuem diferentes formas de ordenação com base em critérios variados de comparação.

Importante

Embora não seja obrigatória, a propriedade de igualdade é fortemente desejável no caso da ordenação natural, isto é, para classes que implementam a interface **Comparable**. Várias classes padrão da biblioteca Java, como **String**, **Integer** ou **Date**, implementam essa interface levando em conta essa característica (ORACLE, 2019).

Por fim, uma propriedade desejável, mas não obrigatória, é a de **igualdade**. Com ela, sempre que houver uma relação `o1.equals(o2)`, a comparação de `o1` e `o2` também deve retornar zero.

4.2

Algoritmos de ordenação



Os algoritmos de ordenação têm como objetivo classificar uma lista inteira e possuem duas operações essenciais (LAFORE, 2005):

- comparar elementos (como vimos anteriormente);
- trocar a posição de elementos fora de ordem.

Iniciaremos pelo estudo de dois algoritmos bastante simples: o bubble sort e o selection sort. Ambos usam uma estratégia similar de varrer todo o vetor de dados. Em seguida, vamos estudar um algoritmo chamado quick sort, que possui uma estratégia um pouco mais avançada e é capaz de subdividir a lista e ordená-la recursivamente.

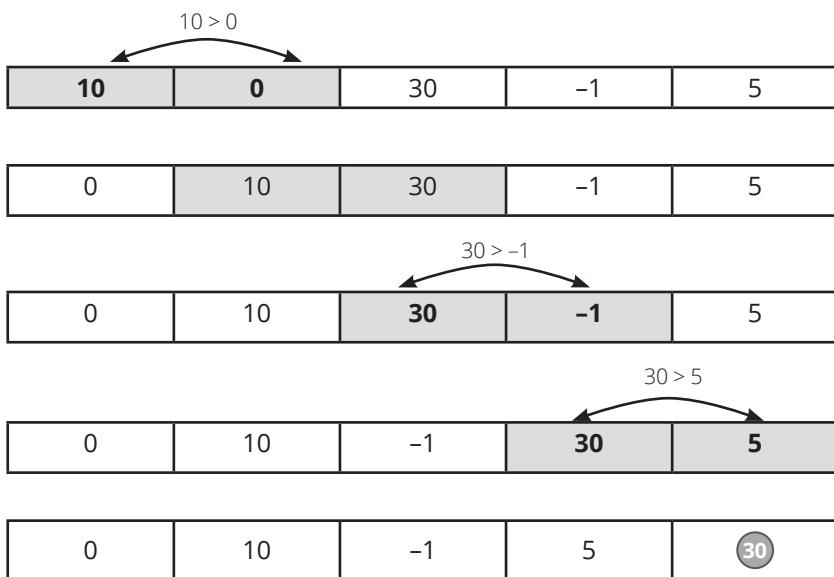
Os algoritmos de ordenação também podem ser classificados como estáveis e não estáveis (GOODRICH; TAMASSIA, 2013). Um algoritmo **estável** garante que, caso dois elementos sejam considerados iguais, eles se mantenham na mesma ordem que estavam antes de a execução ocorrer. Os **não estáveis**, estudados neste capítulo, não dão essa garantia.

4.2.1 Bubble sort

O bubble sort, geralmente traduzido como *algoritmo da bolha*, ordena um vetor percorrendo-o várias vezes (LAFORE, 2005). A cada passada, cada elemento é comparado ao seu vizinho; caso esteja fora de ordem, ele é trocado de lugar. Isso faz com que o maior elemento se desloque para o final do vetor. A figura a seguir mostra uma iteração do algoritmo.



Figura 1
Iteração do bubble sort



Maior elemento

Os valores com fundo cinza representam os elementos sendo comparados. Caso os valores apareçam em negrito, isso significa que o algoritmo identificou que eles devem trocar de lugar.

Fonte: Elaborada pelo autor.

Observe que o último elemento se encontra ordenado. O algoritmo deve repetir o processo novamente, para todos os elementos restantes da lista, até que não haja mais elementos a serem comparados, ou até que a comparação não tenha resultado em nenhuma troca.

Uma das grandes vantagens desse algoritmo é que sua implementação é bastante simples. Por exemplo, esta seria a função `bubbleSort` na classe `ListaSequencial`:

```
1  private void troca(int i, int j) {  
2      var temp = dados[i];  
3      dados[i] = dados[j];  
4      dados[j] = temp;  
5  }  
6  
7  public void bubbleSort(Comparator<? super T> comparator) {  
8      for (var i = 0; i < getTamanho(); i++) {  
9          boolean trocou = false;  
10         for (var j = 0; j < getTamanho()-1-i; j++) {  
11             if (comparator.compare(dados[j], dados[j+1]) > 0) {  
12                 troca(j, j+1);  
13                 trocou = true;  
14             }  
15         }  
16         if (!trocou) {  
17             break;  
18         }  
19     }  
20 }
```

Importante

Observe que, para um vetor de n elementos, o resultado será, no pior caso, $\frac{n(n - 1)}{2}$ comparações e um número equivalente de trocas (MAIN; SAVITCH, 2005).

(Continua)

```

14         }
15     }
16     if (!trocou) break;
17   }
18 }

```

Por ter um alto custo de processamento, esse algoritmo é praticamente inviável para a maioria das aplicações práticas.

4.2.2 Selection sort

No algoritmo *bubble sort*, o número de comparações e trocas é praticamente igual. Em Java, isso não é um grande problema, uma vez que os elementos são objetos e, portanto, conterão apenas referências às áreas de memória em que seus dados estão armazenados (SIERRA; BATES, 2010).

Porém, em outras linguagens, como C#, C++ ou Swift, isso nem sempre será verdade. Essas linguagens permitem definir tipos de dados compostos de valor, como as classes (MAIN; SAVITCH, 2005). Isso significa que os dados podem ficar diretamente colocados no vetor e serem copiados a cada atribuição durante o processo de troca, exatamente como ocorre com tipos primitivos. Um objeto grande, portanto, pode representar um alto custo de performance.

Nesse contexto surge o algoritmo **selection sort**, que apresenta um número de comparações mais alto do que o bubble sort, porém minimiza o número de trocas. Seu processo é bastante simples (LAFORE, 2005):

- 1 o algoritmo começa no primeiro elemento da lista;
- 2 partindo dele, busca-se o menor elemento dentro de toda a lista, comparando-o com os demais;
- 3 ao final da iteração, troca-se o menor elemento da lista com o primeiro;
- 4 o elemento, agora, encontra-se ordenado. Portanto, o algoritmo passa para o próximo elemento da lista e retorna ao passo 2.

Observe que, diferentemente do que ocorre com o *bubble sort*, esse algoritmo não possui uma forma de abandonar a iteração caso o vetor fique ordenado.

Importante

Para um vetor de n elementos, no pior caso, serão feitas $\frac{n^2 - n}{2}$ comparações, mas apenas $(n - 1)$ trocas, pois o último elemento já ficará em seu lugar (MAIN; SAVITCH, 2005).

Na classe `ListaSequencial`, o código do algoritmo se resume em:

```
1  public void selectionSort(Comparator<? super T> comparator) {  
2      for (int i = 0; i < getTamanho()-1; i++) {  
3          int menor = i;  
4          for (int j = i+1; j < getTamanho(); j++) {  
5              if (comparator.compare(dados[menor], dados[j]) > 0) {  
6                  menor = j;  
7              }  
8          }  
9          troca(i, menor);  
10     }  
11 }
```

Uma das vantagens desse algoritmo é que ele também é simples de implementar em uma lista encadeada. Veja a seguir:

```
1  public void selectionSort(Comparator<? super T> comparator) {  
2      var primeiro = this.base;  
3      while (primeiro != null) {  
4          var proximo = primeiro.proximo;  
5          var menor = primeiro;  
6          var elemento = primeiro.proximo;  
7  
8          while (elemento != null) {  
9              if (comparator.compare(menor.dado, elemento.dado) > 0) {  
10                  menor = elemento;  
11              }  
12              elemento = elemento.proximo;  
13          }  
14  
15          var tmp = primeiro.dado;  
16          primeiro.dado = menor.dado;  
17          menor.dado = tmp;  
18  
19          primeiro = proximo;  
20      }  
21 }
```

Observe que o código da linha 15 até 17 está trocando os dados. Uma alternativa para listas encadeadas seria, em vez disso, trocar os nós. Isso é interessante porque implica apenas em atualizar as quatro referências dos nós adjacentes (e o topo e a base, quando for o caso). Assim, garantimos que o custo de troca seja sempre baixo, independentemente do tamanho do dado dentro do nó (MAIN; SAVITCH, 2005).

 Atividade 1

Implemente uma função de troca proposta para a lista encadeada. Tome cuidado, pois um dos nós pode ser o topo ou a base da lista.

4.2.3 Quick sort

Os algoritmos bubble sort e selection sort utilizam a estratégia de “força bruta”, isto é, compararam todos os elementos entre si. Embora essa estratégia seja conceitualmente simples, ela envolve um número significativo de comparações.

Iremos agora analisar uma alternativa muito mais eficiente, baseada na estratégia de “dividir para conquistar”. Trata-se do algoritmo conhecido como **quick sort**, que é dividido em três fases (LAFORE, 2005):

- 1 **Escolher**: escolher um elemento da lista, chamado de *pivô*, que será usado como base nos próximos passos.
- 2 **Separar**: separar a lista, de modo que os elementos inferiores ou iguais ao pivô fiquem à sua esquerda e os elementos maiores que o pivô, à sua direita. O pivô em si não é adicionado a essas listas, pois agora está na sua posição correta.
- 3 **Conquistar**: unir as listas.

O truque desse algoritmo está no fato de ele ser **recursivo**. Observe que, após o passo 2, haverá duas sublistas, uma de cada lado do pivô. Aplica-se a mesma lógica a cada uma delas. Isso prossegue, sucessivamente, até que a lista esteja ordenada. A figura a seguir demonstra esse processo.

 Figura 2

Funcionamento do quick sort

Escolher	30	-1	90	0	10	-7	5
Separar	-1	0	-7	5	30	90	10
Escolher	-1	0	-7	5	30	90	10
Separar	-7	0	-1	5	10	30	90
Escolher	-7	0	-1	5	10	30	90
Separar	-7	-1	0	5	10	30	90
Conquistar	-7	-1	0	5	10	30	90

|| Os valores com fundo cinza são os pivôs a cada passo.

Fonte: Elaborada pelo autor.

Observe que, após a primeira etapa de escolher, a etapa de separar dividiu a nossa lista nas sublistas: à direita do pivô, com os elementos 30, 90 e 10; e à esquerda, com os elementos -7, 0 e 1. O processo, então, repetiu-se para cada uma das próximas sublistas (por isso, a segunda etapa de escolher demonstrada contém dois pivôs).

A segunda etapa de separar geraria quatro sublistas. Porém, como os pivôs escolhidos na lista da esquerda e da direita terminaram na posição inicial da lista, ambas acabaram gerando apenas uma sublista. O processo se repete até que as sublistas tenham tamanho 2. Finalmente, realizou-se a etapa de conquistar, unindo toda as sublistas.

O critério para a escolha do pivô, em nosso exemplo, foi utilizar o último elemento da lista, seja ele qual for. Outros métodos foram propostos para a etapa de escolha do pivô, como a **média dos três**, na qual se obtém o elemento mediano entre o primeiro, o último e o elemento do meio do vetor (MAIN; SAVITCH, 2005). Para se obter a mediana, ordenamos os três elementos e escolhemos o valor do meio. Por exemplo, vamos supor uma lista com 10 elementos. Nela, o elemento de índice 0 tem valor 1, o do índice 5 (meio da lista) tem valor 4 e o do último índice tem valor -7. Ordenamos esses 3, obtendo: -7, 1, 4. Portanto, 1 é o valor mediano entre os três e será o pivô.

No caso das listas sequenciais, o desafio está em criar a função separar de modo que um novo vetor não seja criado e que a posição final do pivô seja conhecida no fim do processo. Isso pode ser feito por meio do algoritmo proposto por Nico Lomuto (BENTLEY, 1999):

- Consideraremos a posição de inserção do menor elemento, que chamaremos de m , como o primeiro elemento da lista (`inicio`).
- Percorremos a lista do primeiro ao penúltimo elemento, pois o último (`fim`) será o pivô. Caso um elemento que está sendo percorrido seja menor ou igual ao pivô, trocamos ele com o de posição m e somamos um em m .
- Ao final do processo, trocamos o último elemento da lista (`pivô`) com o da posição m . O valor de m é retornado.

A figura a seguir mostra o funcionamento da função separar:



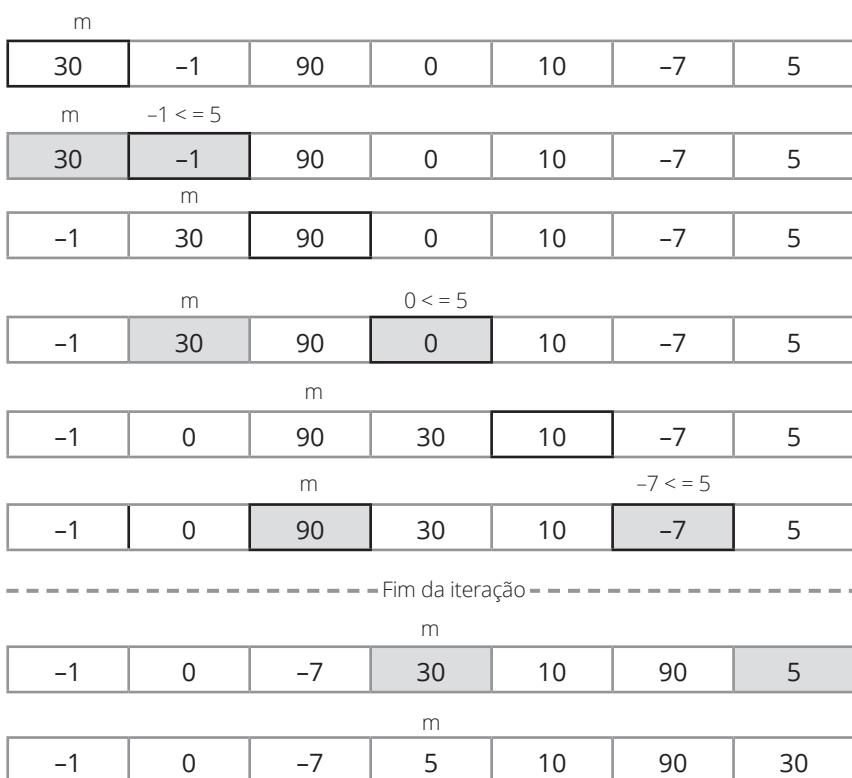
Atenção

No caso de um vetor, como demonstrado na Figura 2, o processo de conquistar não precisa fazer nada. Afinal, apenas os elementos foram movidos. Contudo, veremos que, no caso das listas encadeadas, a implementação desse algoritmo realmente precisará unir listas diferentes.



Figura 3

Funcionamento da função `separar`



O elemento que está sendo percorrido aparece com a borda grossa, em negrito. Os elementos que serão trocados estão com fundo cinza.

Fonte: Elaborada pelo autor.

Podemos alterar a posição de `inicio` e `fim` para separar apenas uma parte da lista. A implementação dessa função é demonstrada a seguir:

```
1  private int separar(Comparator<? super T> comparator, int inicio, int fim) {  
2      var pivo = dados[fim];  
3      var m = inicio;  
4      for (int i = inicio; i < fim; i++) {  
5          if (comparator.compare(dados[i], pivo) <= 0) {  
6              troca(m, i);  
7              m = m + 1;  
8          }  
9      }  
10     //Posiciona o pivo no local correto  
11     troca(m, fim);  
12     return m;  
13 }
```

The screenshot shows a web browser window with the URL <https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/>. The page content discusses the differences between Hoare's and Lomuto partition schemes in Quicksort. At the bottom of the page, there is a timestamp: "Acesso em: 2 dez. 2019."

Para que o quick sort funcione, basta executar três passos (LAFORE, 2005):

- 1 separar a lista, obtendo a posição final do pivô p ;
- 2 executar o quick sort para a sublista de `inicio` até $p - 1$;
- 3 executar o quick sort para a lista de $p + 1$ até `fim`.

Observe que, devido a esses cálculos, a função `quickSort` será chamada com as variáveis `inicio` e `fim` iguais, caso a lista tenha apenas um elemento, ou com `fim` menor que `inicio`, se a lista estiver vazia (casos que representam a condição de parada do algoritmo). Portanto, a implementação final da função será:

```
1 | private void quickSort(Comparator<? super T> comparator, int inicio, int fim) {  
2 |     if (inicio < fim) {  
3 |         var p = separar(comparator, inicio, fim);    // 1  
4 |         quickSort(comparator, inicio, p - 1);      // 2  
5 |         quickSort(comparator, p + 1, fim);        // 3  
6 |     }  
}
```

Por fim, basta criar uma versão pública da função considerando `inicio` e `fim` como a lista inteira:

```
1 | public void quickSort(Comparator<? super T> comparator) {  
2 |     quickSort(comparator, 0, getTamanho() - 1);  
3 | }
```

Vamos agora analisar o quick sort na lista encadeada. Como vimos, a lista encadeada não permite o acesso direto aos índices, por isso seria inviável utilizar o quick sort da forma recém-descrita. Entretanto, uma estratégia bastante simples é utilizar a própria estrutura da lista encadeada para criar as sublistas.



Atividade 2

Tente implementar o método `escolher` usando a lógica da média dos três. Para que a função `separar` não mude, basta trocar o elemento escolhido com o último da lista.

Assim, basta seguirmos os três passos do algoritmo diretamente. Observe:

```
1  public void quickSort(Comparator<? super T> comparator) {  
2      if (getTamanho() < 2) return;  
3  
4      var menores = new ListaEncadeada<T>();  
5      var iguais = new ListaEncadeada<T>();  
6      var maiores = new ListaEncadeada<T>();  
7  
8      var pivo = topo.dado;  
9      for (var dado : this) {  
10          var cmp = comparator.compare(dado, pivo);  
11          if (cmp < 0) {  
12              menores.adicionar(dado);  
13          } else if (cmp == 0) {  
14              iguais.adicionar(dado);  
15          } else {  
16              maiores.adicionar(dado);  
17          }  
18      }  
19  
20      menores.quickSort(comparator);  
21      maiores.quickSort(comparator);  
22  
23      limpar();  
24      adicionarTodos(menores);  
25      adicionarTodos(iguais);  
26      adicionarTodos(maiores);  
27  }
```

O quick sort possui uma vantagem: como as sublistas são realmente distintas, é muito fácil criar uma lista para todos os elementos iguais ao pivô (GOORDRICH; TAMASSIA, 2013). Isso evita que eles sejam reprocessados, já que podemos colocar todos em suas posições finais. Perceba também que a etapa de conquistar agora precisa fundir o conteúdo das listas.

Entretanto, embora a implementação apresentada demonstre o conceito, ela ainda é bastante ineficiente. Cada chamada a `adicionar` cria um novo nó. Além disso, o processo de adição final destrói todos os nós presentes na lista para então recriá-los. Essas operações implicam em custos de alocação e desalocação de memória; se analisarmos o problema com cuidado, veremos que nada disso é necessário. Afinal, o processo de ordenação não modifica os dados e uma implementação eficiente, portanto, reaproveitaria nós.

Para resolver esse problema, vamos criar uma função que “rouba” o primeiro nó de uma lista e o incorpora a outra, chamada `roubarBase`. O primeiro passo para criá-la é fazer uma pequena modificação na função `adicionar`, criando uma versão privada que aceite um nó diretamente:

```
1  private void adicionar(No<T> no) {  
2      if (isVazia()) {  
3          base = no;  
4      } else {  
5          no.anterior = topo; //1  
6          topo.proximo = no; //2  
7      }  
8      topo = no;           //3  
9      tamanho = tamanho + 1; //3  
10 }  
11  
12 @Override  
13 public void adicionar(T valor) {  
14     adicionar(new No<T>(valor));  
15 }
```

Nossa função `remover` antiga não considerava a possibilidade de o nó ser reaproveitado. Por isso, ela não se dava ao trabalho de limpar as referências do nó sendo removido. Como agora isso vai ocorrer, precisamos limpar essas referências antes de remover o nó. Por isso, antes de retornar o dado e subtrair o tamanho, precisamos adicionar essas duas linhas:

```
no.proximo = null;  
no.anterior = null;
```

Depois disso, estamos prontos para criar nossa função `roubarBase`. O processo é simples: basta remover a base da lista recebida por parâmetro e adicioná-la ao final:

```
1 | private void roubarBase(ListaEncadeada<T> outra) {  
2 |     var no = outra.base;  
3 |     outra.remover(no);  
4 |     adicionar(no);  
5 | }
```

Por fim, também precisaremos unir as listas. Para isso, criaremos o método `roubarTodos`, que toma todos os nós da lista recebida por parâmetro:

```
1 | public ListaEncadeada<T> roubarTodos(ListaEncadeada<T> outra) {  
2 |     while (!outra.isVazia()) {  
3 |         roubarBase(outra);  
4 |     }  
5 |     return this;  
6 | }
```

Observe que, ao invés de retornar `void`, o método retorna a própria lista. Essa técnica é chamada de **invocation chaining** (em português: encadeamento de invocações) e facilita o uso de chamadas em sequência, já que agora poderemos unir as três listas com um só comando:



Atividade 3

Crie uma versão mais otimizada do método `roubarTodos`. Observe que, no lugar da iteração, basta unir o topo de uma lista à base da outra e atualizar a variável `tamanho`. Além disso, é importante limpar a lista `outra` ao final do processo.



Desafio

Vários métodos em nossa biblioteca de coleções (que hoje retornam `void`) poderiam ser modificados para suportar a técnica *invocation chaining* (BLOCH, 2019). Que tal revisá-la para adicionar esse suporte onde for possível?

Neste momento, vamos revisar nossa função `quickSort` para utilizar os novos métodos criados. Observe que, como lidaremos com nós, precisaremos iterar sobre a lista nó a nó, e não mais por meio do comando `for each`:

```

1  public void quickSort(Comparator<? super T> comparator) {
2      if (getTamanho() < 2) return;
3
4      //Separar
5      var menores = new ListaEncadeada<T>();
6      var iguais = new ListaEncadeada<T>();
7      var maiores = new ListaEncadeada<T>();
8
9      var pivo = topo.dado;
10     var it = base;
11     while (it != null) {
12         var prox = it.proximo;
13         var cmp = comparator.compare(it.dado, pivo);
14         if (cmp < 0) {
15             menores.roubarBase(this);
16         } else if (cmp == 0) {
17             iguais.roubarBase(this);
18         } else {
19             maiores.roubarBase(this);
20         }
21         it = prox;
22     }
23
24     //Conquistar
25     menores.quickSort(comparator);
26     maiores.quickSort(comparator);
27     roubarTodos(menores).roubarTodos(iguais).roubarTodos(maiores);
28 }
```

Observe que as listas menores, iguais e maiores “roubam” o nó da nossa lista e, ao final do processo, ela “rouba de volta” todos os nós das sublistas. Assim, os nós são apenas transferidos de uma lista para outra, sem que seja feita nenhuma alocação ou desalocação adicional. Observe também que, na linha 12, tomamos o cuidado de criar uma variável auxiliar para armazenar o próximo nó da lista. Isso ocorre pois `no.proximo` irá se tornar nulo assim que o nó for roubado, já que esse nó se tornará o topo de sua nova lista.

Importante

A performance do método **quickSort** varia de acordo com a escolha do pivô. O pior caso, quando a lista está ordenada, decairá para uma performance quadrática, similar à dos métodos anteriores. Entretanto, o melhor caso, com o pivô perfeitamente balanceado, obterá uma performance para n elementos na ordem de $n \log(n)$, o que torna o algoritmo viável para aplicações reais (MAIN; SAVITCH, 2005).

Artigo

<https://pt.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort>

O artigo *Análise do quicksort*, publicado no site do Khan Academy, demonstra, de maneira detalhada e com gráficos, um processo de análise do quick sort. Assim, você entenderá como é calculado o custo desse algoritmo no pior e no melhor caso e em uma execução normal. A leitura desse texto é indicada para que você aprenda um pouco mais sobre como fazemos a crítica de algoritmos mais complexos.

Acesso em: 2 dez. 2019.

As bibliotecas comerciais implementam outras otimizações, como processos eficientes para a escolha do pivô e para os algoritmos de separação; uso de algoritmos de ordenação mais simples, quando a lista se torna pequena; redução do tamanho da recursão do algoritmo; entre outras.

4.3

Listas ordenadas

 Videoaula



Organizar os elementos não é a única forma de garantir a ordem. Uma alternativa a esse processo é sempre inserir os elementos no local correto. Nesta seção, discutiremos o conceito de *lista ordenada*.

Embora já entendamos a teoria por trás disso, a abstração desse conceito em uma classe será tema apenas do próximo capítulo. Por enquanto, iremos apenas criar métodos na classe `ListaSequencial` que permitam a inserção e busca de elementos de forma ordenada. Obviamente, esses métodos só podem ser usados em conjunto; por isso, não serão mantidos na versão final da nossa classe.

4.3.1 Buscando elementos

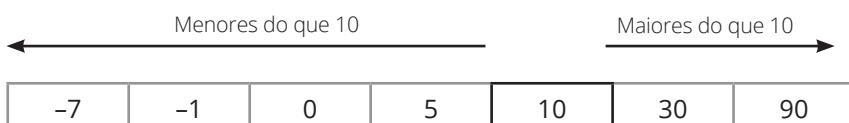
O primeiro passo para criar uma lista ordenada é implementar a função `indice`. Lembre-se de que essa função busca um elemento e retorna a posição dele na lista, ou `-1`, se ele não for encontrado. No caso desse tipo de lista, essa função tem um papel adicional: ela deve retornar o **ponto de inserção**, caso o elemento não seja encontrado (LAFORE, 2005). Isto é, caso um elemento não exista, ela deve indicar qual seria o índice em que esse elemento deveria ser inserido na lista.

Em nossa função índice da lista sequencial, iterávamos sobre a lista toda até encontrar o elemento. Entretanto, no caso de uma lista ordenada, há uma abordagem muito mais inteligente, conhecida como **busca binária**. Ela parte da observação de que, dado um elemento qualquer, todos os elementos à sua esquerda são menores do que ele e todos os elementos à sua direita, maiores. Observe a seguir:



Figura 4

Ordem dos elementos na lista ordenada



Fonte: Elaborada pelo autor.

Para buscar um valor, podemos, então, utilizar a abordagem a seguir (LAFORE, 2005).

- 1 Pegamos o elemento do meio da lista. Então, há três possibilidades:
 - a Ele pode ser igual ao que buscamos. Nesse caso, a busca terminou.
 - b Ele pode ser **maior** do que o valor que estamos buscando. Nesse caso, descartamos todos os elementos à direita dele.
 - c Ele pode ser **menor** do que o elemento que estamos buscando. Nesse caso, descartamos os elementos à esquerda dele.
- 2 Após o descarte, repetimos o passo 1 na sublista formada pelos elementos não descartados.

A figura a seguir ilustra esse processo para uma lista de 15 elementos.



Figura 5

Busca binária do valor 100

39 < 100															
Início															Fim
-30	-18	-10	-8	2	10	21	39	52	100	101	200	587	1004	9810	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	

200 > 100															Fim
Início															Fim
-30	-18	-10	-8	2	10	21	39	52	100	101	200	587	1004	9810	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	

100 = 100															Fim
Início															Fim
-30	-18	-10	-8	2	10	21	39	52	100	101	200	587	1004	9810	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	

A borda forte, em negrito, indica o elemento do meio, que está sendo analisado pela busca. Os valores com fundo cinza indicam os elementos já descartados no processo.

Fonte: Elaborada pelo autor.

Observe que essa estratégia permite que o número de elementos buscados aumente de maneira logarítmica em relação ao tamanho da lista. Assim, para realizarmos um simples passo a mais na busca, a lista precisaria dobrar de tamanho.

Vejamos o que ocorre quando um elemento não existe.

 Figura 6

Busca do valor inexistente 9

Início															Fim
-30	-18	-10	-8	2	10	21	39	52	100	101	200	587	1004	9810	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
-8 < 9															Fim
-30	-18	-10	-8	2	10	21	39	52	100	101	200	587	1004	9810	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
10 > 9															Fim
-30	-18	-10	-8	2	10	21	39	52	100	101	200	587	1004	9810	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
2 < 9															Início Fim
-30	-18	-10	-8	2	10	21	39	52	100	101	200	587	1004	9810	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
Fim Início															
-30	-18	-10	-8	2	10	21	39	52	100	101	200	587	1004	9810	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	

Fonte: Elaborada pelo autor.

1

Para manter consistência com nossa biblioteca, um nome melhor seria *indiceBinario*, mas manteremos o nome clássico do algoritmo (*buscaBinária* ou *pesquisaBinária*) para facilitar seu estudo.

Observe que, no penúltimo passo, *inicio* e *fim* estão posicionados no mesmo elemento. Assim, ou ele será menor do que o elemento desejado, fazendo com que *inicio* avance, ou será maior, fazendo com que o *fim* recue, gerando a condição de término da busca: *inicio > fim*. Além disso, observe que o valor de *inicio* estará no ponto de inserção, que seria o índice 5, no caso do valor 9. Para permitir que o usuário saiba desse ponto, nossa função retornará esse valor negativado e subtraído de um, ou seja -6. Por que subtrair um? Porque o ponto de inserção poderia ser o 0, que não possui um número negativo associado.

Vamos criar a função *buscaBinaria*¹ que retorna esse índice:

```

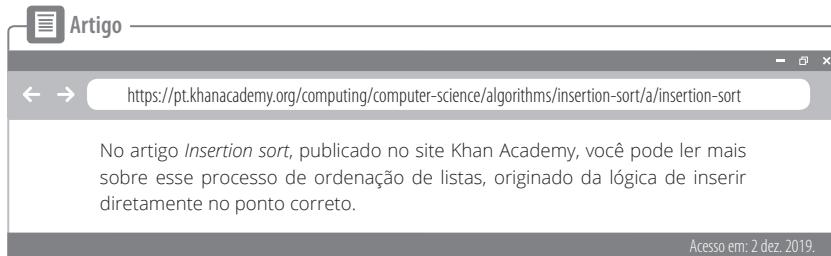
1  public int buscaBinaria(Comparator<? super T> comparator, T dado) {
2      int inicio = 0;
3      int fim = getTamanho()-1;
4      while (inicio <= fim) {
5          int meio = (inicio + fim) / 2;
6          T dadoDoMeio = dados[meio];
7
8          int cmp = comparator.compare(dadoDoMeio, dado);
9
10         if (cmp == 0) {
11             return meio;                                //Encontrado
12         } else if (cmp < 0) {
13             inicio = meio + 1;                         //Descarta a esquerda
14         } else if (cmp > 0) {
15             fim = meio - 1;                           //Descarta a direita
16         }
17     }
18     return -(inicio+1);                            //Não encontrado
19 }
```

Agora, podemos acrescentar o método `adicionarEmOrdem`, que adicionará o elemento em sua posição correta. Esse método não permitirá a inserção de elementos duplicados:

```

1  public boolean adicionarEmOrdem(Comparator<? super T> comparator, T dado) {
2      int ind = buscaBinaria(comparator, dado);
3
4      //Elemento duplicado
5      if (ind > 0) {
6          return false;
7      }
8
9      //Insere o elemento
10     int ponto = -(ind+1);
11     adicionar(ponto, dado);
12     return true;
13 }
14 }
```

Observe que a mesma lógica pode ser usada no processo de remoção e que também poderíamos utilizar a busca binária para criar uma versão otimizada do método contem.



A screenshot of a web browser window titled "Artigo". The URL in the address bar is <https://pt.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort>. The page content discusses the insertion sort algorithm, mentioning its origin from the logic of inserting directly into the correct point. At the bottom right of the page, it says "Acesso em: 2 dez. 2019."

E a lista encadeada? Lembre-se de que ela não permite a obtenção de elementos intermediários diretamente, o que inviabiliza o processo de busca binária (MAIN; SAVITCH, 2005). Para resolver esse problema, utilizamos outra estrutura de dados, conhecida como *árvore binária*, que será estudada no último capítulo deste livro.

Por fim, note que os métodos `adicionarOrdenado` e `buscaBinaria`, como parte dos métodos da lista, simplesmente não funcionam se a lista não estiver ordenada. Por isso, muitas bibliotecas, como a do próprio Java, não incluem esses métodos como parte das listas, e sim como métodos utilitários (SIERRA; BATES, 2010). Entretanto, esses conceitos podem ser usados na criação de outras estruturas de dados, como os conjuntos ordenados, que veremos nos próximos capítulos.



CONSIDERAÇÕES FINAIS

Neste capítulo, começamos a explorar o conceito de *dados ordenados*. Obviamente, eles não são os únicos algoritmos existentes. Uma rápida busca na internet nos revela nomes como merge sort, heap sort ou comb sort, além de um número muito grande de variações desses algoritmos, que buscam melhorias de *performance* ou garantia de estabilidade na ordenação. Entretanto, o grupo de algoritmos que estudamos nos permite ter facilidade no estudo desses outros, pois já podemos traçar paralelos e entender os compromissos envolvidos ao pensarmos em ordenação.

Nos capítulos seguintes, vamos estudar mais abordagens para organizar dados com base em suas características, dando a você mais munição para entender o funcionamento de vários sistemas e bibliotecas que utilizará no futuro.



REFERÊNCIAS

- BENTLEY, J. *Programming Pearls*. 2. ed. Boston: Addison-Wesley Professional, 1999.
- BLOCH, J. *Java Efetivo: as melhores práticas para a plataforma Java*. 3. ed. Rio de Janeiro: Alta Books, 2019.
- DEITEL, P.; DEITEL, H. *Java: como programar*. 8. ed. São Paulo: Pearson do Brasil, 2009.
- GOODRICH, M. T.; TAMASSIA, R. *Estruturas de Dados & Algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.
- LAFORE, R. *Estruturas de dados & algoritmos em Java*. 2. ed. Rio de Janeiro: Ciência Moderna, 2005.
- MAIN, M.; SAVITCH, W. *Data structures & Other objects using C++*. 3. ed. Boston, MS: Pearson, 2005.
- ORACLE. *Interface Comparable*, 2019. Disponível em: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Comparable.html>. Acesso em: 2 dez. 2019.
- SIERRA, K.; BATES, B. *Use a cabeça, Java!* Rio de Janeiro: Alta Books, 2010.



GABARITO

- Dividimos o problema em duas funções. A primeira, chamada de `atualizaAdjacentes`, recebe como parâmetro o nó e os dois nós que serão seus novos vizinhos. A segunda é a função `troca`, que chama a função anterior para os dois nós que estão sendo trocados. Assim:

```
1  private void atualizaAdjacentes(No<T> no, No<T> novoAnterior, No<T>
2      novoProximo) {
3      no.anterior = novoAnterior;
4      if (novoAnterior == null) {
5          base = no;
6      } else {
7          novoAnterior.proximo = no;
8      }
9
10     no.proximo = novoProximo;
11     if (novoProximo == null) {
12         topo = no;
13     } else {
14         novoProximo.anterior = no;
15     }
16 }
17
18 private void troca(No<T> no1, No<T> no2) {
19     if (no1 == no2) return;
20 }
```

```

21   var anteriorNo1 = no1.anterior;
22   var proximoNo1 = no1.proximo;
23
24   var anteriorNo2 = no2.anterior;
25   var proximoNo2 = no2.proximo;
26
27   atualizaAdjacentes(no1, anteriorNo2, proximoNo2);
28   atualizaAdjacentes(no2, anteriorNo1, proximoNo1);
29 }

```

2. Observe que nossa solução implementa manualmente a mesma lógica do bubble sort para os elementos `dados[inicio]`, `dados[fim]` e `dados[meio]`, nessa ordem. Dessa forma, `dados[fim]` conterá o valor de pivô intermediário entre os três:

```

1  private T escolher(Comparator<? super T> comparator, int inicio, int fim) {
2      if (fim - inicio > 3) {
3          int meio = (fim + inicio) / 2;
4
5          if (comparator.compare(dados[inicio], dados[fim]) > 0) {
6              troca(inicio, fim);
7          }
8
9          if (comparator.compare(dados[fim], dados[meio]) > 0) {
10             troca(fim, meio);
11         }
12
13         if (comparator.compare(dados[inicio], dados[fim]) > 0) {
14             troca(inicio, fim);
15         }
16     }
17
18     return dados[fim];
19 }

```

Para usar a função, basta alterar o início da função `separar` para:

```
var pivo = escolher(comparator, inicio, fim);
```

3. Nossa implementação foi baseada na do método `adicionar`. Assim, ela simplesmente une o topo da nossa lista à base da lista recebida por parâmetro. Após isso, todo o tamanho da lista é somado:

```
1 private ListaEncadeada<T> roubarTodos(ListaEncadeada<T> outra) {
2     if (outra.isVazia()) return this;
3
4     var no = outra.base;
5     if (isVazia()) {
6         base = no;
7     } else {
8         no.anterior = topo;
9         topo.proximo = no;
10    }
11    topo = outra.topo;
12    tamanho += outra.tamanho;
13    outra.limpar();
14
15    return this;
16}
17 }
```

Espalhamento

Estudamos, com as coleções ordenadas, formas bastante eficientes de encontrar dados. Porém, uma boa biblioteca de coleções também deve fornecer mecanismos velozes para a localização de dados não ordenados. Uma dessas alternativas é utilizar a estratégia de espalhamento, que permite associar dados entre si.

Neste capítulo, vamos entender como podemos implementar estruturas de dados de modo eficiente com essa estratégia.

5.1 Conceitos



Vamos fazer uma analogia para entender o conceito de **espalhamento**, também conhecido como *dispersão* ou *hashing*. Vamos supor que você tenha uma coleção qualquer, como de selos ou bonecas, e queira organizá-la de modo a encontrar rapidamente um dos elementos. No local onde os objetos serão guardados, existe um conjunto de gavetas. Você poderia optar por um critério, como tamanho, e agrupar objetos que atendam a esse critério.

O que aconteceria se você precisasse procurar um dos itens da sua coleção? Por meio do critério selecionado, você encontraria a gaveta adequada e, após isso, a abriria e procuraria um a um até encontrar o item desejado.

E como podemos mapear o conceito do gaveteiro para a computação? Fazemos isso por meio de tabelas e funções de espalhamento.



MyVisuals/Shutterstock

5.1.1 Tabelas de espalhamento

Na computação, essa estratégia do gaveteiro é conhecida como **tabela de espalhamento**. Com ela, podemos implementar duas estruturas de dados:

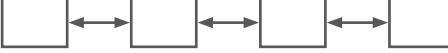
- **mapas**: nos permitem associar dois objetos entre si, como o número da matrícula (chave) ao objeto que representa um funcionário (valor).
- **conjuntos**: contém uma sequência sem repetição de objetos. Eles nos permitem, por exemplo, gerar uma lista contendo todos os remetentes de um grupo grande de e-mails, sem repeti-los.

Como você já pode ter deduzido, o gaveteiro é uma analogia para um vetor, em que cada índice representa uma gaveta. Como cada gaveta terá vários objetos similares, precisaremos inserir uma lista em cada uma de suas posições (GOODRICH; TAMASSIA, 2013). Essa estrutura, como um todo, é chamada de **tabela hash**.



Figura 2

Tabela hash: vetor de listas contendo objetos.

0	
1	
2	
3	
4	
5	

Fonte: Elaborada pelo autor.

Agora, precisamos criar uma forma inteligente de determinar o índice da lista em que ele será inserido, com base em um objeto qualquer. Por exemplo, suponhamos a classe `Livro`, descrita a seguir.

```

1  public class Livro {
2      private String nome;
3      private int paginas;
4      private String autor;
5
6      public Livro(String nome, int paginas, String autor) {
7          this.nome = (nome == null ? "" : nome);
8          this.paginas = paginas;
9          this.autor = autor;
10     }
11
12     public String getNome() {
13         return nome;
14     }
15
16     public int getPaginas() {
17         return paginas;
18     }
19
20     public String getAutor() {
21         return autor;
22     }
23
24     @Override
25     public boolean equals(Object o) {
26         if (this == o) return true;
27         if (o == null || o instanceof Livro) return false;
28         Livro livro = (Livro) o;
29         return paginas == livro.paginas &&
30             nome.equals(livro.nome);
31     }
32 }
```

Observe que o critério de igualdade de dois livros (definido no método `equals` dessa classe) é que o nome e o número de páginas devem ser iguais.

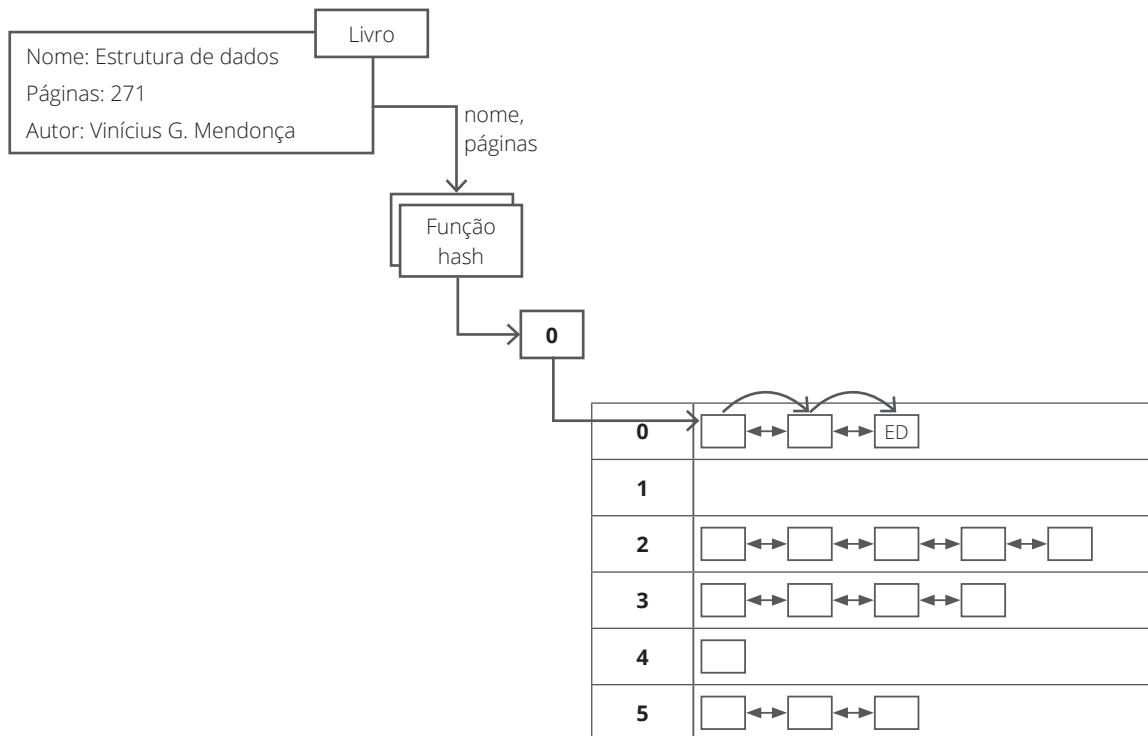
E como poderíamos colocar vários livros de uma biblioteca no vetor de nossa tabela de espalhamento de modo a rapidamente localizar se um livro está ou não presente em seu interior? Fazemos isso por meio

de uma função conhecida como **função de espalhamento** ou **função hash**. Ela tem como objetivo converter um conjunto qualquer de valores em um outro valor, chamado **código hash**, que estará dentro de um intervalo numérico conhecido (LAFORE, 2005).

Por exemplo, poderíamos computar os dados do nome e do número de páginas e gerar um número no intervalo de zero até o tamanho do vetor subtraído de um. Esse valor nos indicaria diretamente a lista em que o objeto poderia ser encontrado. Então, bastaria buscá-lo nessa lista.

Observe que, para essa estratégia funcionar, é importante que a função hash sempre retorne o mesmo número para dois objetos iguais. Além disso, para que o processo seja eficiente, é importante que o código gerado tenha como tendência gerar valores bem distribuídos entre todos os índices disponíveis, pois, caso contrário, poderemos sobrecarregar apenas algumas posições do vetor, enquanto outras ficam ociosas (GOODRICH; TAMASSIA, 2013). A figura a seguir ilustra o processo completo.

 **Figura 3**
Como encontrar um elemento em uma tabela hash.



Fonte: Elaborada pelo autor.

 **Vídeo**

As funções hash não são usadas somente para a geração de tabelas. Elas também são úteis em bibliotecas de compressão, validação de dados e criptografia. O vídeo *Criptografia – Hashes – Resumos de Mensagens - 09*, do canal Bóscon Treinamentos, explica como essa função é usada e quais são as boas características dela no contexto da criptografia.

Disponível em: <https://www.youtube.com/watch?v=f8arNurR6rk>. Acesso em: 7 jan. 2020.

Como o número de índices no vetor é finito e, geralmente, muito menor do que o número de elementos que serão armazenados em seu interior, ocorrerão **colisões de códigos hash**. A Figura 3 também ilustra essa situação: o código hash para o livro *Estruturas de dados* computado em nosso exemplo foi 0. Isso fez com que fosse localizada a lista correspondente no índice 0 do vetor. Devido à colisão de hashes, já havia três livros cadastrados, que precisaram ser testados um a um para verificar se um deles era o desejado. No caso do nosso exemplo, esse livro era o último elemento da lista.

Curiosidade

Em português, a tradução literal de bucket é balde. Porém, no contexto da informática, não possuímos uma tradução direta para o termo e, por isso, optamos por manter o nome em inglês.

As listas em uma tabela hash têm um nome especial: **bucket**. Observe que, para ter boa performance, dependemos de dois fatores:

- A função hash precisa ser veloz e bem distribuída; caso contrário, perderíamos muito tempo no seu cálculo ou acabaríamos gerando buckets muito longos, que demorariam para ser percorridos.
- A quantidade de buckets deve permitir que eles não contenham muitos elementos, já que, em seu interior, a busca é linear. A razão entre a quantidade real de elementos (tamanho) e a quantidade de buckets é chamada de **carga**. Por um lado, quando o valor da carga é alto, a busca é demorada, pois temos listas cheias. Por outro, um valor muito baixo pode gerar desperdício de memória, uma vez que temos um grande número de listas completamente vazias. No caso do Java, considera-se que um fator de carga de 0,75 como um bom compromisso entre os dois lados (ORACLE, 2019a).

Obviamente, um mapa dinâmico irá alterar de tamanho e redistribuir os objetos sempre que precisar. Essa operação é chamada de **rehash** e, similarmente ao que ocorre nas listas sequenciais, tem um custo alto, visto que precisará repositionar todos os elementos.

5.1.2 A função hashCode

As funções hash são tão importantes que, assim como no caso da função `equals` (que testa se dois objetos são iguais), possuímos na classe `Object` a função `hashCode`, que deve retornar um número inteiro com código hash de dois objetos (SIERRA; BATES, 2010).

Há várias implementações possíveis para essa função. Por exemplo, caso o dado seja uma chave numérica única (como um número de matrícula), uma implementação trivial possível seria retornar esse valor

diretamente. Essa função não só tem custo zero, como também é considerada perfeita, uma vez que cada valor será mapeado para apenas um único resultado (MAIN; SAVITCH, 2005).

Entretanto, para dados mais complexos, nos quais múltiplos campos precisam ser combinados (caso da classe Livro usada em nosso exemplo), temos que usar uma estratégia mais sofisticada. Uma abordagem possível é utilizar a estratégia proposta por Bloch (2019):

- 1 Comece por um valor constante armazenado em uma variável chamada `result`, por exemplo, o valor 79.
- 2 Para cada campo `f` utilizado no método `equals` em seu objeto, calcule o valor `c` com base nas seguintes regras:
 - a. Se o campo for booleano, calcule: `f ? 1 : 0`;
 - b. Se o campo for dos tipos byte, char, short ou int, seu valor será diretamente: `(int) f`;
 - c. Se o campo for do tipo long, calcule: `(int) (f ^ (f >> 32))`. Isso fará com que os 32 bits iniciais do long sejam combinados com os 32 bits finais, resultando em um valor inteiro de 32 bits;
 - d. Caso o valor seja um float, utilize a função `Float.floatToIntBits`. Essa função retorna o valor dos bits da variável diretamente, na forma de um número inteiro;
 - e. Caso o número seja um double, utilize a função `Double.doubleToLongBits` e aplique ao resultado a fórmula descrita em 2c;
 - f. Caso o dado seja um objeto, use: `f == null ? 0 : f.hashCode()`. Não será necessário testar se `f` é nulo caso o campo não admita esse campo;
 - g. Caso o elemento seja um vetor, trate-o como se cada valor dentro do vetor fosse um campo separado aplicando a ele as regras descritas anteriormente.
- 3 Combine o valor `c`, calculado nos campos da etapa 2, dentro da variável `result` por meio da fórmula: `result = p * result + c`, em que `p` é um número primo qualquer (como o valor 31).

Observe que a função utiliza uma série de operações simples e muito velozes, por contar praticamente só com somas, multiplicações



Vídeo

Você sabia que os dois últimos números do seu CPF também são uma função hash simples, conhecida como dígitos verificadores ou, simplesmente, CRC (*cyclic redundancy check*)? Por meio deles, é possível verificar se o número está correto, evitando que um CPF inválido seja inserido acidentalmente em cadastros. Você pode aprender a calcular seu CPF no vídeo *Como calcular os dígitos verificadores do CPF - Minuto Educação #165*, do canal Planneta Educação.

Disponível em: <https://www.youtube.com/watch?v=15Bw0duulMQ>. Acesso em: 7 jan. 2020.

e operações de bit. Ela também garante uma boa distribuição dos resultados dentro dos valores possíveis para o tipo `int`. Finalmente, cabe destacar que a função pode considerar menos campos do que os presentes no `equals`, caso sejam redundantes (BLOCH, 2019).

No caso da classe `Livro`, por exemplo, o algoritmo seria implementado da seguinte maneira:

```
1 public int hashCode() {  
2     int result = 79;  
3     result = 31 * result + (nome == null ? 0 : nome.hashCode());  
4     result = 31 * result + (int) paginas;  
5     return result;  
6 }
```

Essa implementação segue rigorosamente o que foi descrito, mas ela pode ser simplificada. Analise que, pela implementação da classe `Livro`, o campo `nome` nunca será nulo; por isso, não precisaríamos testar essa condição. Além disso, não precisamos fazer um cast para `int` da variável `paginas`, visto que ela já possui esse mesmo tipo de dado. Assim, isso resultaria na função:

```
1 public int hashCode() {  
2     int result = 79;  
3     result = 31 * result + nome.hashCode();  
4     result = 31 * result + paginas;  
5     return result;  
6 }
```

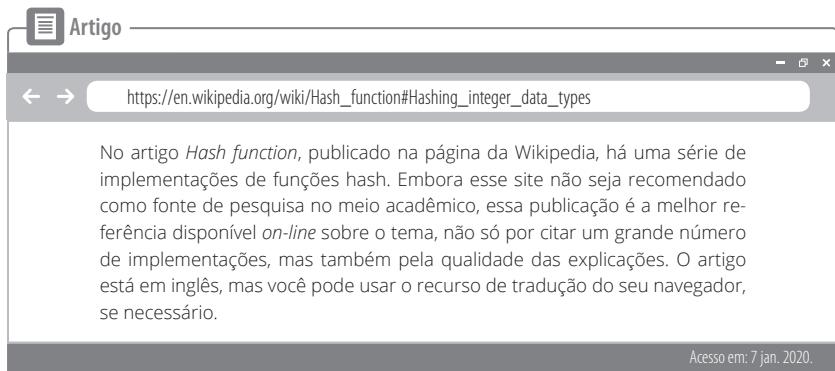
Por fim, essa implementação é tão comum que o Java possui a função `Objects.hash`, que realiza esse cálculo automaticamente:

```
1 @Override  
2 public int hashCode() {  
3     return Objects.hash(nome, paginas);  
4 }
```

Logicamente, há várias outras implementações possíveis para a função `hashCode`, mas essa já é suficiente para nossos estudos. As demais implementações serão apenas variações na forma de combinar os dados.



A função `Objects.hash` foi incluída na versão 7 do Java e fornece uma assinatura mais conveniente à função `Arrays.hashCode`, presente na plataforma desde a versão 5.



A screenshot of a web browser window titled "Artigo". The address bar shows the URL https://en.wikipedia.org/wiki/Hash_function#Hashing_integer_data_types. The main content area contains text about hash functions, mentioning that the article is in English but can be translated. At the bottom right of the content area, there is a timestamp "Acesso em: 7 jan. 2020."

No artigo *Hash function*, publicado na página da Wikipedia, há uma série de implementações de funções hash. Embora esse site não seja recomendado como fonte de pesquisa no meio acadêmico, essa publicação é a melhor referência disponível *on-line* sobre o tema, não só por citar um grande número de implementações, mas também pela qualidade das explicações. O artigo está em inglês, mas você pode usar o recurso de tradução do seu navegador, se necessário.

Acesso em: 7 jan. 2020.

5.1.3 Endereçamento aberto

Utilizar uma lista em cada posição de um vetor não é a única forma de se implementar uma tabela de espalhamento. Na tentativa de poupar espaço em memória, podemos utilizar a estratégia de **endereçamento aberto**, em que tentamos manter todos os dados apenas no vetor principal (LAFORE, 2004).

Uma opção de endereçamento aberto é a utilização de uma estratégia chamada de *endereçamento duplo*, em que utilizamos duas funções hash. Aplicamos a primeira função e, caso uma colisão ocorra, utilizamos a função secundária. Caso haja colisão novamente, podemos fazer o rehash, ampliando o vetor principal.

Outra opção é procurar um índice livre do vetor linearmente. Para isso, fazemos o cálculo com a função hash e, caso o índice já esteja ocupado, verificamos o seu vizinho. Caso continue ocupado, passamos para o próximo índice, até que um índice livre seja encontrado. Uma das dificuldades adicionais dessa abordagem está na exclusão de itens, pois é necessário verificar todos os vizinhos, caso haja a necessidade de movimentação. Isso pode ser atenuado utilizando-se um valor especial como marcador de *disponível* em cada casa vazia.

Esses tipos de uso só se justificam em dispositivos que a quantidade de memória é criticamente baixa (MAIN; SAVITCH, 2005). Afinal, o overhead provocado por uma lista encadeada em cada item não é tão grande ao ponto de justificar uma implementação mais complexa. Além disso, as listas também apresentam boas características de performance para a inclusão e exclusão de elementos.

5.1.4 Interface Mapa

A estratégia descrita até aqui para as tabelas de espalhamento é suficiente para a implementação de conjuntos, isto é, verificar se um dado está ou não presente na lista e evitar sua duplicação. Porém, não é incomum que dados possam ser associados por meio de um valor especial, conhecido como *chave*.

Uma **chave** é um valor sem repetição que permite identificar unicamente o elemento que queremos buscar (GOODRICH; TAMASSIA, 2013). Por exemplo, no Brasil, as pessoas são identificadas pelo cadastro de pessoas físicas (CPF), que é um número único para cada cidadão. As **estruturas associativas**, como os vetores e os mapas, permitem que, de posse de uma chave, um objeto seja rapidamente localizado.

Contudo, diferentemente do que ocorre em um vetor, cuja chave está limitada a um índice, um mapa permitirá que suas chaves tenham descontinuidades ou não sejam sequer numéricas (ORACLE, 2019a).

A implementação de um mapa é similar ao que já descrevemos até aqui. A diferença é que ela insere, em cada valor da tabela hash, uma estrutura conhecida como **entrada**, que nada mais é do que um par de valores relacionados (LAFORE, 2004). Um dos valores é a chave, que identifica o objeto, onde faremos todas as operações hash e buscas já descritas, e o outro valor é o objeto associado a essa chave. Podemos implementar a classe dos pares de valores como uma classe interna estática da interface mapa, conforme o código a seguir.

```
1  public interface Mapa<C, V> {
2      class Par<C, V> {
3          private C chave;
4          private V valor;
5
6          public Par(C chave, V valor) {
7              this.chave = chave;
8              this.valor = valor;
9          }
10
11         public C getChave() {
12             return chave;
13         }
14     }
```

(Continua)

```

15     public V getValor() {
16         return valor;
17     }
18
19     protected V setValor(V valor) {
20         var antigo = this.valor;
21         this.valor = valor;
22         return antigo;
23     }
24
25     @Override
26     public String toString() {
27         return chave + ":" + valor;
28     }
29
30 }
```

Note que o método `setValor` tem visibilidade `protected`. Isso permite que nossa implementação modifique valores associados diretamente à chave, mas não o usuário de nossa classe.

Já a interface Mapa precisa conter os métodos que associam a chave C aos valores V. Embora essa interface não seja filha de Colecao, implementaremos uma série de métodos similares, como o método `limpar`, ou análogos – o método `contem`, por exemplo –, que, nesse caso, busca pela chave em vez do objeto inteiro:

```

1 public interface Mapa<C, V> {
2     //A CLASSE PAR VAI AQUI
3
4     //Informações
5     int getTamanho();
6     boolean isEmpty();
7
8     //Inclusão de itens
9     V adicionar(C chave, V valor);
10
11    //Exclusão de itens
12    void limpar();
13    V remover(C chave);
```

(Continua)

```

15 //Acesso direto a valores
16 V get(C chave);
17
18 //Métodos para buscar itens
19 boolean contem(C chave);
20
21 //Iteração
22 Iterator<Par<C, V>> iterator();
23 }

```

Outro ponto que deve ter chamado sua atenção é que a interface Mapa, por si só, não é iterável. Isso porque a iteração do mapa pode ocorrer de três formas (ORACLE, 2019a): pelas chaves, pelos valores ou pelos pares ordenados (chave/valor). Forneceremos três métodos padrão diferentes para isso neste capítulo. Entretanto, temos, na interface, o método `iterator`, que será utilizado como base para construir esses três tipos de iteráveis.

5.1.5 Interface Conjunto

Os conjuntos possuem uma estrutura mais simples. Eles são unicamente uma coleção que não permite elementos duplicados (ORACLE, 2019b). Além de todos os métodos de coleção, os conjuntos possuirão os métodos `contem`, `remover` e `getTamanho`:

```

1 public interface Conjunto<T> extends Colecao<T> {
2     boolean contem(Object valor);
3     int getTamanho();
4     boolean remover(T valor);
5 }

```

Verifique que os conjuntos podem ser facilmente implementados na forma de mapas, em que os elementos associados às chaves são todos nulos. Utilizaremos, neste capítulo, o padrão de projetos *adapter* (GAMMA *et al.*, 2007) para criar uma classe de conjuntos que pode ser utilizada com qualquer tipo de mapa, inclusive os que serão vistos no próximo capítulo.

5.2 Implementação do mapa não ordenado —



Vamos criar a primeira implementação de nosso mapa utilizando a estratégia de espalhamento. Para tanto, teremos três variáveis básicas:

- 1 O vetor buckets. Cada bucket nada mais é do que uma lista encadeada de objetos da classe `Mapa.Par`.
- 2 A variável `tamanho`, que representa a quantidade de elementos efetivamente inseridos no mapa.
- 3 Uma variável chamada `fatorCarga`. Ela representa a carga máxima suportada pelo mapa antes que uma operação de `rehash` seja feita.

Também, forneceremos dois construtores para a classe: um que permite especificar os valores tanto do fator de carga quanto do número inicial de buckets; e um segundo, mais conveniente, que usa os valores padrão 0,75 e 16 (valores que também são usados na API Java). Essas implementações são descritas pelo código a seguir.

```
1  public class MapaHash<C, V> implements Mapa<C, V> {  
2      private ListaEncadeada<Par<C, V>>[] buckets;  
3      private int tamanho;  
4      private double fatorCarga;  
5  
6      public MapaHash() {  
7          this(16, 0.75);  
8      }  
9  
10     public MapaHash(int nrBuckets, double fatorCarga) {  
11         this.fatorCarga = fatorCarga;  
12         buckets = criarBuckets(nrBuckets);  
13     }  
14  
15     protected ListaEncadeada<Par<C, V>>[] criarBuckets(int tamanho) {  
16         var buckets = new ListaEncadeada[tamanho];  
17         for (var i = 0; i < buckets.length; i++) {  
18             buckets[i] = new ListaEncadeada<>();  
19         }  
20         return buckets;  
21     }  
22 }
```



Importante

Estamos utilizando, por conveniência, a classe `ListaEncadeada` criada nos capítulos anteriores. Porém, nossa implementação utiliza uma lista duplamente encadeada. Uma alternativa mais eficiente seria utilizar uma lista com encadeamento simples, similar a uma `Fila`.

Finalmente, note que também fornecemos a função `criarBuckets`. Ela criará o vetor necessário para a tabela hash, além de todas as listas encadeadas em cada uma de suas posições. Inicialmente, todas as listas estarão vazias. Essa função foi separada dessa forma porque ela também será necessária na operação de `rehash`.

5.2.1 Informações do mapa

Todos os mapas fornecem seu tamanho e indicam se está vazio ou não, como informações básicas. A implementação desses dois métodos é similar ao que já fizemos para as listas. Além dessas duas informações, os mapas com base em hash também informam a sua carga atual, definida pela divisão da quantidade de elementos pela quantidade de buckets. A seguir, são descritos os métodos de informação básicos.

```
1  @Override
2  public int getTamanho() {
3      return tamanho;
4  }
5
6  @Override
7  public boolean isVazio() {
8      return tamanho == 0;
9  }
10
11 public double getCarga() {
12     return (double) tamanho / buckets.length;
13 }
14
15 public double getFatorCarga() {
16     return fatorCarga;
17 }
```



Atenção

Não confunda o **fator de carga**, que é a carga máxima suportada por esse mapa antes da operação de `rehash`, com a **carga** propriamente dita. Enquanto a primeira é um valor fixo (como 0,75), a segunda será recalculada a cada elemento inserido.

5.2.2 Localizando itens

Um fator crítico em praticamente todos os métodos do mapa está em localizar o item na tabela hash. Mesmo no caso de uma inclusão, precisamos nos certificar de que o elemento não existe, uma vez que um mapa não permite duplicações de elementos com a mesma chave.

Para tanto, vamos criar uma estrutura auxiliar chamada **Localizacao**, que nos indica:

- o bucket onde o item está, ou deveria estar;
- um iterador, já posicionado sobre o elemento encontrado;
- uma estrutura **Par**, contendo a chave e o valor encontrados.

Caso o elemento não esteja no mapa, tanto o iterador quanto a estrutura do par *chave/valor* serão nulos. Como essa classe foi criada apenas para facilitar nossa implementação do mapa, ela será mantida como uma classe interna privada da classe **MapaHash**:

```
1 | private static class Localizacao<C, V> {
2 |     private ListaEncadeada<Par<C, V>> bucket;
3 |     private Iterator<Par<C, V>> iterator;
4 |     private Par<C, V> par;
5 |
6 |     public boolean naoAchou() { return par == null; }
7 |     public V getValor() { return par == null ? null : par.getValor(); }
8 |     public void setValor(V valor) { return par.setValor(valor); }
9 |     public void remover() { iterator.remove(); }
10 |    public void adicionar(C chave, V valor) {
11 |        bucket.adicionar(new Par<>(chave, valor));
12 |    }
13 | }
```

Atente para essa estrutura que também contém alguns métodos utilitários, apresentados no Quadro 1.

 **Quadro 1**
Métodos utilitários da classe **Localizacao**

Método	Descrição
naoAchou	Retorna verdadeiro, caso o par não seja encontrado.
getValor	Retorna o valor encontrado ou nulo, caso o valor não seja encontrado.
setValor	Substitui o valor encontrado, retornando antigo valor.
remover	Remove o elemento encontrado do bucket onde ele está. Essa operação é possível pois, dentro do objeto Localizacao , está armazenado o iterador do bucket correto, já na posição onde o elemento foi encontrado.
adicionar	Cria um par <i>chave/valor</i> e o adiciona ao final do bucket.

Fonte: Elaborado pelo autor.

Agora, vamos implementar a função `acharPar`, que será responsável por preencher o objeto da classe `Localizacao` com os dados de onde o par foi encontrado. Para que isso seja possível, ela deverá:

- 1 calcular o código hash da chave chamando a função `hashCode`;
- 2 reduzir o código hash para um valor no intervalo dentro dos índices disponíveis no vetor `buckets`;
- 3 localizar o bucket referente ao código hash calculado;
- 4 iterar sobre ele em busca da chave desejada;
- 5 construir o objeto do tipo `Localizacao`, que descreve o resultado dos itens acima.

Você deve estar se perguntando sobre o passo 2 dessa lista. A implementação do método `hashCode`, de acordo com o que foi especificado na plataforma Java, gera um código hash para um número inteiro qualquer¹. Esse valor deve ser mapeado para o intervalo de zero até o tamanho do vetor `buckets` subtraído de um, que é o intervalo de índices disponíveis nesse vetor. Podemos fazer isso por meio de duas operações:

- 1 Calcular o resto entre o número gerado e o tamanho do vetor. O resto pode ser obtido pelo operador `%`.
- 2 Usar a função `Math.abs` para garantir que o resultado da divisão seja sempre positivo. Por exemplo, caso o vetor tenha tamanho 5 e o número gerado seja 18, o resto dessa divisão será 3. Agora, se o número gerado fosse negativo, como -18, o resto seria -3.

Criaremos, para isso, a função estática auxiliar `reduzir`, descrita a seguir.

```
1 | private static int reduzir(int hash, int tamanho) {  
2 |     return Math.abs(hash % tamanho);  
3 | }
```

Com esses elementos em mãos, podemos criar a função `acharNo`, descrita a seguir.

```
1 | public Localizacao<C, V> acharPar(C chave) {  
2 |     var local = new Localizacao<C, V>();  
3 |  
4 |     //Obtemos o bucket  
5 |     int idx = reduzir(chave.hashCode(), buckets.length);  
6 |     local.bucket = buckets[idx];
```

(Continua)

1

Um número inteiro é qualquer valor de $-2.147.483.648 (-2^{31})$ até $2.147.483.647 (2^{31}-1)$, visto que o tipo inteiro representa uma variável numérica de 32 bits com sinal (ORACLE, 2019c).

Importante

A função `reduzir` não é a única forma de reduzir um código hash. Caso o vetor `buckets` sempre possuísse um tamanho múltiplo de 2, seria possível utilizar a operação de máscara de bits por meio do operador inteiro E (`&`), que é ainda mais rápido. A operação seria resumida a: `return hash & (buckets.length-1)`.

```

7
8     //Iteramos em busca da chave
9     var it = local.bucket.iterator();
10    while (it.hasNext()) {
11        var par = it.next();
12
13        //Caso a chave seja encontrada atualizamos
14        //os dados de iterator e par
15        if (par.getChave().equals(chave)) {
16            local.iterator = it;
17            local.par = par;
18            break;
19        }
20    }
21    return local;
22 }

```

Agora que já somos capazes de localizar o ponto exato onde nosso objeto está (ou deveria estar), vamos às operações básicas.

5.2.3 Adição de elementos

Adicionamos um elemento fornecendo a chave e o valor para o qual aquela chave será mapeada. Caso uma chave seja fornecida mais de uma vez, o valor mapeado será substituído e retornado pelo método adicionar. Para realizar essa operação, precisaremos pesquisar se já não há um valor mapeado para aquela chave e

1 Caso não encontremos:

- adicionamos o valor à lista e testamos se o mapa agora não excede o fator de carga máximo permitido;
- caso exceda, precisaremos fazer o rehash. Em ambos os casos, o método retorna nulo, já que não havia valor mapeado para aquela chave.

2 Caso encontremos, basta substituir o valor pelo fornecido, retornando o valor antigo. Note que o método setValor das classes Par e Localizacao já se comportam dessa forma.

O código completo é descrito a seguir.



Importante

É perfeitamente válido associar um valor nulo a uma chave. Nesse caso, será impossível distinguir, pelo retorno do método **adicionar**, se o valor nulo foi substituído ou não. Caso nosso usuário precise dessa distinção, será necessário utilizar o método **contem**.

```

1  @Override
2  public V adicionar(C chave, V valor) {
3      var local = acharPar(chave);
4      if (local.naoAchou()) {
5          local.adicionar(chave, valor);
6          tamanho = tamanho + 1;
7
8          if (getCarga() > fatorCarga) {
9              rehash();
10         }
11         return null;
12     }
13
14     return local.setValor(valor);
15 }
```

Atividade 1

Na função `reduzir`, recalculamos o `hashCode` de cada chave novamente.

Uma alternativa para isso seria gastarmos um pouco mais de memória para armazenar o hash da chave dentro da estrutura do par, evitando recalculá-lo sempre que o rehash fosse feito.

Implemente essa alternativa.

E como fazemos a operação de rehash? Criaremos um novo vetor `buckets`, com o dobro de tamanho, e recolocaremos nele todos os elementos do vetor atual:

```

1  private void rehash() {
2      var novosBuckets = criarBuckets(buckets.length * 2);
3
4      for (var par : entradas()) {
5          var idx = reduzir(
6              par.getChave().hashCode(), novosBuckets.length);
7          novosBuckets[idx].adicionar(par);
8      }
9
10     buckets = novosBuckets;
11 }
```

Em nossa implementação, utilizamos um iterador que nos permite percorrer o mapa par a par chamado `entradas`. Sua construção será explicada na Subseção 5.2.6.

Desafio

Que tal tentar alterar a visibilidade da função `roubarNo` da `ListaEncadeada` para `protected` e gerar uma versão otimizada da função `rehash` que não recria nós?

5.2.4 Removendo elementos

A operação de limpeza de um mapa é bastante simples. Basta percorrer o `buckets` limpando todas as listas ali presentes:

```

1  @Override
2  public void limpar() {
3      for (var bucket : buckets) {
4          bucket.limpar();;
5      }
6  }

```

Para remover um único elemento, basta chamar o método remove do iterador do bucket em que ele foi localizado (essa é exatamente a implementação do método remover da classe Localizacao) e, em seguida, reduzir o tamanho do mapa em um. Caso o elemento não esteja no mapa, o método de exclusão simplesmente retornará nulo:

```

1  @@Override
2  public V remover(C chave) {
3      //Tentamos localizar o par a ser removido
4      var local = acharPar(chave);
5
6      //Se não achar, retorna nulo
7      if (local.naoAchou()) {
8          return null;
9      }
10
11     //Remove da lista onde foi encontrado
12     local.remover();
13     tamanho = tamanho - 1;
14
15     //Retorna o valor recém removido
16     return local.getValor();
17 }

```

5.2.5 Acesso direto a valores

O acesso direto a valores é feito por meio do método get, que é implementado diretamente com a função acharNo. O mesmo vale para a função contem:

```

1  @Override
2  public V get(C chave) {
3      return acharPar(chave).getValor();
4  }
5

```

(Continua)

```

6 |     @Override
7 |     public boolean contem(C chave) {
8 |         return !acharPar(chave).naoAchou();
9 |

```

Lembre-se de que a função `contem` é a única forma inequívoca de testar se uma chave está realmente presente no mapa. Entretanto, na prática, os valores de retorno das demais funções geralmente são suficientes, pois raramente mapeamos valores nulos.

5.2.6 Iteração

Repare que nossa interface Mapa possui o método `iterator`, mas não implementa a interface `Iterable`. A razão disso é que existem três formas de iterar sobre o mapa e, por isso, criaremos métodos padrão para cada iteração diferente, chamados `entradas`, `valores` e `chaves`. Nossa objetivo é permitir um método de iteração similar ao `for` a seguir.

```
for (var valor : mapa.chaves()) {
```

Vamos começar analisando o iterador do mapa. Na classe desse iterador, vamos criar três variáveis: a variável `i`, que irá controlar em qual lista estamos percorrendo; a variável `iterator`, que irá conter o iterador da lista atual; e a variável `cont`, que contará quantos elementos já foram percorridos:

```

1 |     private class IteradorMapa implements Iterator<Par<C, V>> {
2 |         int i = -1;
3 |         int cont = 0;
4 |         Iterator<Par<C, V>> iterator = null

```

Observe que o valor inicial de `i` é `-1`, pois o iterador inicia posicionando antes do primeiro elemento. O método `hasNext` é bastante simples e, com ele, podemos verificar o valor de `cont` contra o tamanho do mapa:

```

1 |     @Override
2 |     public boolean hasNext() {
3 |         return cont < tamanho;
4 |     }

```

Já o método `next` funciona da seguinte forma:

- 1 Caso ele já esteja iterando por uma lista, avança um passo no iterador da lista e retorna o elemento iterado.
- 2 Caso o iterador tenha acabado ou a iteração não tenha começado, busca-se uma lista com elementos. Ao encontrá-la, inicializa-se a variável `iterator` com seu valor e retorna seu primeiro elemento.

O seguinte código mostra essas operações:

```

1  @Override
2  public Par<C, V> next() {
3      if (!hasNext()) throw new NoSuchElementException();
4
5      //Passo 1: Caso já haja um iterador com elementos
6      //avança e retorna o elemento
7      if (iterator != null && iterator.hasNext()) {
8          cont = cont + 1;
9          return iterator.next();
10     }
11
12     //Passo 2: Busca uma lista com elementos
13     while (buckets[++i].isVazia());
14
15     //Atualiza o iterador e retorna seu primeiro
16     //elemento
17     iterator = buckets[i].iterator();
18     cont = cont + 1;
19     return iterator.next();
20 }
```

A remoção também é um processo simples. Como estamos utilizando os iteradores das listas, basta chamarmos o método `remove` e atualizar o tamanho do mapa e o contador:

```

1  @Override
2  public void remove() {
3      iterator.remove();
4      tamanho = tamanho - 1;
5      cont = cont - 1;
6  }
```

Agora, basta retornarmos esse valor no método `iterator` da nossa classe do MapaHash:

```

1 |     @Override
2 |     public Iterator<Par<C, V>> iterator() {
3 |         return new IteradorMapa();
4 |     }

```

Então, podemos criar os métodos padrão para a iteração na interface Mapa. Eles funcionarão para qualquer mapa que a implemente, desde que o método `iterator` seja implementado, como fizemos.

Para isso, iremos primeiro utilizar o padrão de projetos adapter (GAMMA *et al.*, 2007) para gerar iteradores mais simples que retornam somente a chave ou somente o valor. Essas classes utilizarão de base o iterador de pares. A seguir, apresentamos o exemplo da classe adaptadora para as chaves:

```

1 | class ChaveIterator<C, V> implements Iterator<C> {
2 |     private Iterator<Mapa.Par<C, V>> iterator;
3 |
4 |     public ChaveIterator(Iterator<Mapa.Par<C, V>> iterator) {
5 |         this.iterator = iterator;
6 |     }
7 |
8 |     @Override
9 |     public boolean hasNext() {
10 |         return iterator.hasNext();
11 |     }
12 |
13 |     @Override
14 |     public C next() {
15 |         return iterator.next().getChave();
16 |     }
17 |
18 |     @Override
19 |     public void remove() {
20 |         iterator.remove();
21 |     }
22 |

```

Agora, basta implementarmos os métodos padrão propriamente ditos. Eles devem retornar um objeto `Iterable` que retorne esses ite-

radores. Felizmente, utilizando o recurso de lambda, suas implementações ficam muito simples:

```
1 //Iteração
2 default Iterable<C> chaves() {
3     return () -> new ChaveIterator<>(iterator());
4 }
5
6 default Iterable<V> valores() {
7     return () -> new ValorIterator<>(iterator());
8 }
9
10 default Iterable<Par<C, V>> entradas() {
11     return this::iterator;
12 }
13
```

Por fim, vamos também criar um método chamado `forEach`, que permitirá iterar facilmente sobre o mapa.

```
1 default void forEach(BiConsumer<C, V> consumer) {
2     for (var par : entradas()) {
3         consumer.accept(par.getChave(), par.getValor());
4     }
5 }
```

A classe `BiConsumer` só define uma interface cujo método `accept` aceita dois valores (a chave e o valor) e faz o que quiser com eles. Ela é uma das classes padrão do pacote `java.util.function`. Com ela, também podemos iterar sobre o mapa da seguinte forma:

```
mapa.forEach((c, v) ->
    System.out.println("A chave é " + c + " e o valor é " + v));
```



Atividade 2

Escreva uma função que teste o mapa, chamando os métodos `get` e `contém`, além de iterar por todos os valores e observar como é a ordem de impressão dos elementos.

5.2.7 Implementação dos conjuntos

Um conjunto é um grupo de elementos sem repetição. Se você prestar atenção, essa é uma característica das chaves dos mapas. Por isso, muitas bibliotecas, inclusive a do Java, optam por implementar conjuntos por meio de mapas cujos valores sempre serão mapeados para nulo. Essa será nossa implementação também. Similarmente ao que fizemos com os iteradores da classe Mapa, ire-



Atividade 3

É possível substituir o `Iterable` de retorno da função `chaves` do mapa por um `ConjuntoAdapter`. Faça essa substituição e teste o que ocorre ao adicionar elementos utilizando o conjunto retornado.

mos criar uma classe adaptadora que nos permitirá tratar qualquer mapa como se fosse um conjunto.

Sua implementação é muito simples, visto que se resumirá a uma chamada ao respectivo método na classe mapa:

```
1  public class ConjuntoAdapter<T> implements Conjunto<T> {
2      private Mapa<T, ?> mapa;
3
4      public ConjuntoAdapter(Mapa<T, ?> mapa) {
5          this.mapa = mapa;
6      }
7      @Override
8      public void limpar() {
9          mapa.limpar();
10     }
11
12     @Override
13     public void adicionar(T valor) {
14         mapa.adicionar(valor, null);
15     }
16
17     @Override
18     public void remover(T valor) {
19         mapa.remover(valor);
20     }
21
22     @Override
23     public boolean isVazia() {
24         return mapa.isVazio();
25     }
26
27     @Override
28     public boolean isCheia() {
29         return false;
30     }
31 }
```

(Continua)

```

31
32     @Override
33     public boolean contem(Object valor) {
34         return mapa.contem((T)valor);
35     }
36
37     @Override
38     public int getTamanho() {
39         return mapa.getTamanho();
40     }
41
42     @Override
43     public Iterator<T> iterator() {
44         return mapa.chaves().iterator();
45     }
46 }
```

Observe que essa implementação usa como base a interface `Mapa`, e não a classe `MapaHash`. Isso significa que ela poderá ser usada inclusive para os mapas que criaremos no próximo capítulo.

Desafio

Tente copiar a classe `MapaHash` e criar a classe `ConjuntoHash`. Em seguida, ajuste o código para a classe ficar mais enxuta, eliminando a estrutura auxiliar `Par`.



CONSIDERAÇÕES FINAIS

Neste capítulo, exploramos o poder das tabelas de espalhamento e das funções hash. Combinadas, elas nos permitiram implementar mapas e conjuntos que funcionam com objetos não ordenados.

Apesar de o processo de localizar um elemento parecer um tanto intrincado, ele garante um tempo de acesso praticamente constante e pequeno o suficiente para que essas estruturas sejam úteis para uma série de propósitos.

Não é à toa que as classes `mapa` e `conjunto`, vistas neste capítulo, também encontram similares na biblioteca de coleções da plataforma Java, sendo conhecidas por lá como `HashMap` e `HashSet`, respectivamente.

Obviamente, caso os objetos no interior dessas estruturas pudessem ser ordenados, também seria possível implementar mapas e conjuntos que se beneficiassem dessa propriedade. Esse será o tema do nosso próximo capítulo, em que conheceremos as árvores.



REFERÊNCIAS

- BLOCH, J. *Java Efetivo*: as melhores práticas para a plataforma Java. 3 ed. Rio de Janeiro: Alta Books, 2019.
- GAMMA, E. et al. *Padrões de Projeto*: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2007.
- GOODRICH, M. T.; TAMASSIA, R. *Estruturas de Dados & Algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.
- LAFORE, R. *Estruturas de dados & algoritmos em Java*. 2. ed. Rio de Janeiro: Ciência Moderna, 2005.
- MAIN, M.; SAVITCH, W. *Data structures & Other objects using C++*. 3. ed. Boston: Pearson, 2005.
- ORACLE. *Interface Map<K,V>*, 2019a. Disponível em: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Map.html>. Acesso em: 18 dez. 2019.
- ORACLE. *Interface Set<E>*, 2019b. Disponível em: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Set.html>. Acesso em: 18 dez. 2019.
- ORACLE. Primitive Data Types. *The Java Tutorials*, 2019c. Disponível em: <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>. Acesso em: 18 dez. 2019.
- SIERRA, K.; BATES, B. *Use a cabeça, Java!* Rio de Janeiro: Alta Books, 2010.



GABARITO

- Para armazenar o hashCode, podemos modificar a classe Par ou criar uma subclasse. Embora a primeira implementação seja mais simples, ela também traria impacto em outros mapas que não utilizassem a estratégia de hash. Por isso, apresentaremos a segunda alternativa.

O primeiro passo é criar a classe ParHash, subclasse de Par:

```
1 class ParHash<C, V> extends Mapa.Par<C, V> {  
2     private int hash;  
3  
4     public ParHash(C chave, V valor) {  
5         super(chave, valor);  
6         this.hash = chave.hashCode();  
7     }  
8  
9     public int getHash() {  
10        return hash;  
11    }  
12}
```

Em seguida, alteramos o método adicionar da classe Localizacao para que ele use o ParHash:

```
1 public void adicionar(C chave, V valor) {  
2     bucket.adicionar(new ParHash<>(chave, valor));  
3 }
```

Finalmente, alteramos a função rehash para usar nossa nova estrutura. Como o iterador ainda retorna a interface Par, é necessário fazer um cast na linha 5:

```
1 | private void rehash() {  
2 |     var novosBuckets = criarBuckets(buckets.length * 2);  
3 |  
4 |     for (var par : entradas()) {  
5 |         var parHash = (ParHash<C, V>)par;  
6 |         var idx = reduzir(parHash.getHash(), novosBuckets.length);  
7 |         novosBuckets[idx].adicionar(par);  
8 |     }  
9 |  
10 |     buckets = novosBuckets;  
11 | }
```

2. Vários programas podem ser feitos para testar o mapa. A seguir, apresentamos um exemplo.

```
1 | public class Main {  
2 |     public static void main(String[] args) {  
3 |         Mapa<String, Integer> mapa = new MapaHash<>();  
4 |  
5 |         //Adicionamos alguns elementos  
6 |         mapa.adicionar("Banana", 1);  
7 |         mapa.adicionar("Laranja", 2);  
8 |         mapa.adicionar("Amora", 3);  
9 |         mapa.adicionar("Caju", 4);  
10 |        mapa.adicionar("Uva", 5);  
11 |        mapa.adicionar("Cereja", 6);  
12 |  
13 |        System.out.println();  
14 |        System.out.println("Imprimindo todos os elementos");  
15 |        mapa.entradas().forEach(System.out::println);  
16 |        System.out.println("Tamanho " + mapa.getTamanho());  
17 |  
18 |        System.out.println("O valor de laranja é " + mapa.get("Laranja"));  
19 |        System.out.println("Tem amora? " + mapa.contem("Amora"));  
20 |        System.out.println("Tem morango? " + mapa.contem("Morango"));
```

```
21 //Removendo os elementos iniciados com "C"
22 var it = mapa.chaves().iterator();
23 while (it.hasNext()) {
24     var chave = it.next();
25     if (chave.startsWith("C")) {
26         it.remove();
27     }
28 }
29
30 var antigo = mapa.adicionar("Uva", 1000);
31 System.out.println("Antigo valor da uva: " + antigo);
32 System.out.println();
33 System.out.println("Imprimindo todos os elementos");
34 mapa.entradas().forEach(System.out::println);
35 System.out.println("Tamanho: " + mapa.getTamanho());
36 }
37 }
```

Esse programa imprimirá a seguinte saída:

Imprimindo todos os elementos

Uva: 5
Laranja: 2
Amora: 3
Banana: 1
Caju: 4
Cereja: 6
Tamanho 6
O valor de laranja é 2
Tem amora? true
Tem morango? false
Antigo valor da uva:5

Imprimindo todos os elementos
Uva: 1000
Laranja: 2
Amora: 3
Banana: 1
Tamanho: 4

Observe que as frutas, ao serem impressas, não seguem qualquer ordem aparente, isto é, não respeitam a ordem de inserção ou a ordem alfabética. Essa é uma das características de nossa implementação. A ordem pode, até mesmo, mudar ao longo do tempo, caso mais elementos sejam inseridos e um rehash ocorra.

3. O primeiro passo para a modificação proposta por esse exercício é excluir a classe ChaveAdapter, que não será mais necessária. Agora, modificamos o método padrão chaves da interface Mapa para:

```
1 | default ConjuntoAdapter<C> chaves() {  
2 |     return new ConjuntoAdapter<C>(this);  
3 | }
```

Finalmente, vamos escrever uma função de teste. Observe que, como agora as chaves são um conjunto, podemos utilizar as funções adicionar e remover diretamente:

```
1 | public class Main {  
2 |     public static void main(String[] args) {  
3 |         Mapa<String, Integer> mapa = new MapaHash<>();  
4 |         //Adicionamos alguns elementos  
5 |  
6 |         mapa.adicionar("Banana", 1);  
7 |         mapa.adicionar("Laranja", 2);  
8 |         mapa.adicionar("Amora", 3);  
9 |         mapa.adicionar("Uva", 4);  
10 |  
11 |         var chaves = mapa.chaves();  
12 |         chaves.adicionar("Cereja");  
13 |         chaves.remover("Laranja");  
14 |  
15 |         System.out.println("Imprimindo");  
16 |         mapa.entradas().forEach(System.out::println);  
17 |     }  
18 | }
```

O código imprime:

```
Imprimindo  
Uva: 4  
Amora: 3  
Banana: 1  
Cereja: null
```

Note que o valor `null` foi associado à cereja, uma vez que ela foi adicionada utilizando o conjunto.

Árvores binárias

No Capítulo 4, aprendemos que dados podem ser ou não ordenados. Além disso, vimos que os processos de busca em dados ordenados são extremamente eficientes, devido ao algoritmo de busca binária, que nos permite descartar rapidamente boa parte dos dados, caso o elemento central seja maior, menor ou igual ao que está sendo procurado.

Entretanto, para obter máxima eficiência, a busca binária possui um requisito: os dados devem ser diretamente acessíveis, visto que o elemento central deverá ser consultado a todo momento. Por causa dessa característica, sua eficiência é grande em estruturas sequenciais, mas fica bastante comprometida em estruturas encadeadas.

Neste capítulo, veremos a estrutura da árvore binária de busca, que é encadeada e garante, ao mesmo tempo, a ordenação natural de seus elementos. Também, estudaremos alguns outros tipos de árvore e seus objetivos.

6.1

Conceitos



Muitas vezes, os dados são ordenados de maneira hierárquica, isto é, de modo que um elemento esteja associado a um conjunto de elementos subordinados. Atribuímos o nome de **árvore** a essa organização (GOODRICH; TAMASSIA, 2013). Um exemplo bastante conhecido dessa estrutura é a estrutura de pastas, presentes no disco. A estrutura a seguir descreve a organização deste livro.

Cada item nessa estrutura é chamado de **nó**. Note que cada item pode ter um conjunto de **nós filhos**, que estão diretamente associados a ele, mas cada um deles deve conter um único **nó pai**. Em nosso exemplo, isso significa dizer que um arquivo não pode estar simultaneamente em duas pastas. Além disso, não pode conter ciclos, com um nó filho apontando para qualquer um de seus nós ascendentes, como pai, avô etc. (LAFORE, 2005).

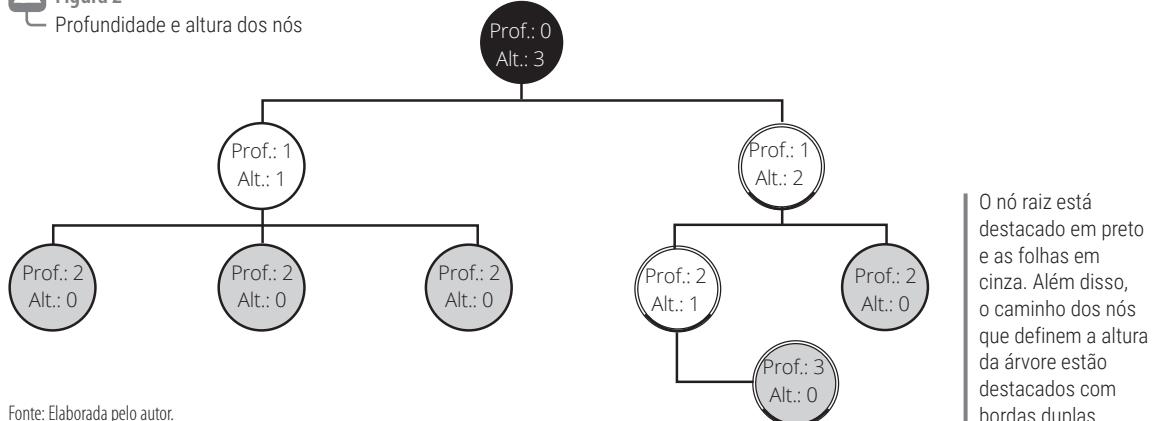
Observe que as árvores possuem um nó inicial, no caso do nosso exemplo, o da pasta *Estrutura de dados*. Ele é denominado **nó raiz**. Além disso, alguns não possuem filhos e, por essa razão, são chamados de **nós folhas** (GOODRICH; TAMASSIA, 2013). No caso do exemplo, os arquivos representam os nós folhas da árvore de pastas. Além disso, observe que uma pasta vazia também seria uma folha, caso existisse.

Dois conceitos importantes em uma árvore são os conceitos de profundidade e altura de um nó (MAIN; SAVITCH, 2005):

- **Profundidade:** é a quantidade de nós entre o nó atual e a raiz, que tem profundidade 0.
- **Altura:** é a quantidade de nós entre o nó sendo analisado e a folha, considerando o caminho mais longo possível. A altura do nó raiz é considerada a altura da árvore, que também é igual à profundidade do nó mais profundo.

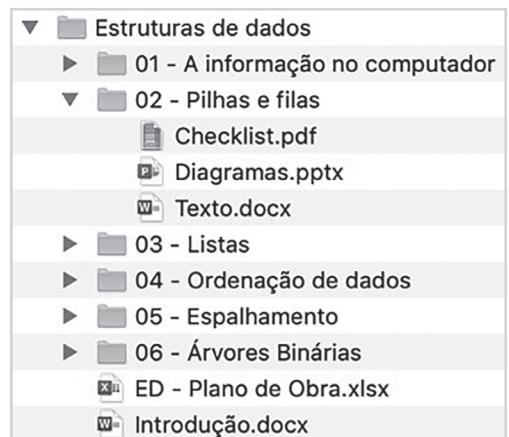
A figura a seguir ilustra esses conceitos.

 Figura 2
Profundidade e altura dos nós



Fonte: Elaborada pelo autor.

 Figura 1
Organização de pastas deste livro



Fonte: Elaborada pelo autor.

Observe que nós em um mesmo nível podem ter alturas diferentes, visto que seus filhos podem ter profundidades diferentes.

6.2

Implementando um mapa ordenado

 Videoaula



Agora, vamos utilizar uma árvore para fazer a implementação de um mapa ordenado. Como você deve se lembrar, mapas são estruturas de dados que associam chaves a valores (ORACLE, 2019a). No caso de um mapa ordenado, as chaves serão únicas e ordenadas, de acordo com um critério de comparação definido.

Para que nossa implementação seja possível, não utilizaremos uma árvore qualquer, e sim uma **árvore binária de busca**. Trata-se de uma árvore com as seguintes limitações (LAFORE, 2005):

- cada nó pode conter apenas dois nós filhos, que chamaremos de **nó esquerdo** e **nó direito**;
- valores menores ou iguais ao nó atual devem ser inseridos ao lado esquerdo. Já os maiores, à direita.

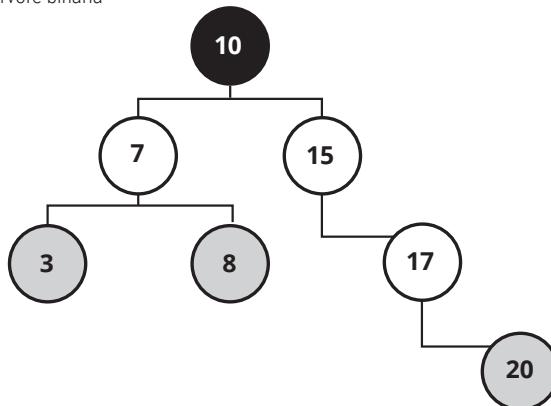
A seguir, ilustramos uma árvore binária, em que os elementos 10, 7, 3, 8, 15, 17 e 20 são inseridos nessa ordem.

Importante

As árvores binárias de busca também são chamadas de BST. Essa sigla vem de seu nome em inglês binary search tree.

 Figura 3

Exemplo de árvore binária



Fonte: Elaborada pelo autor.

Atividade 1

Vamos demonstrar a importância da ordem de inserção. Redesenhe a árvore da Figura 3 considerando que os elementos foram inseridos na seguinte ordem:

- 8, 3, 7, 20, 15, 17, 10
- 3, 7, 8, 10, 15, 17, 20

Essa organização tem por objetivo reduzir o tempo de busca. Perceba que os dados “se dividem” de maneira similar ao que ocorre na busca binária. De fato, uma árvore em que os nós se encontrem perfeitamente distribuídos terá um tempo de busca logarítmico. Entretanto, a disposição dos nós é diretamente dependente da ordem de inserção dos dados (GOODRICH; TAMASSIA, 2013).

6.2.1 A classe ArvoreBinaria

Antes de analisar a classe da árvore binária, vamos entender a estrutura do nó. Assim como fizemos nos capítulos anteriores, essa classe será implementada na forma de uma classe estática interna privada, no interior da classe ArvoreBinaria.

O nó armazenará o par chave/valor (mesma classe vista no Capítulo 5) e possuirá uma referência ao nó esquerdo e nó direito. Além disso, o nó conterá o método `isFolha`, que retorna verdadeiro caso ele não possua nós filhos:

```
1  public class ArvoreBinaria<C, V> implements Mapa<C, V> {
2      private static class No<C, V> {
3          Par<C, V> par;
4          No<C, V> esquerda;
5          No<C, V> direita;
6
7          No(C c, V v) {
8              this.par = new Par<>(c, v);
9          }
10
11         boolean isFolha() {
12             return esquerda == null && direita == null;
13         }
14     }
```

Agora, vamos à classe da árvore em si. Observe que ela será utilizada para implementar um mapa ordenado e, por isso, ela implementa a interface `Mapa`. Como as árvores binárias precisam ordenar as chaves, é necessário referenciar também um objeto comparador, capaz de comparar duas chaves entre si. Além disso, a árvore terá uma referência para seu nó raiz e uma variável que indica seu tamanho, isto é, a quantidade de nós em seu interior:

```
1  public class ArvoreBinaria<C, V> implements Mapa<C, V> {
2      private Comparator<C> comparator;
3      private No<C, V> raiz;
4      private int tamanho = 0;
5
6      public ArvoreBinaria(Comparator<C> comparator) {
7          this.comparator = comparator;
8      }
```



Vídeo

Além de buscas, árvores binárias podem ser utilizadas para uma série de aplicações. Uma delas é a compressão de dados. No vídeo Codificação de Huffman, elaborado e publicado pelo autor Vinícius Godoy de Mendonça, apresentamos o processo de Codificação de Huffman, demonstrando como ele funciona.

Disponível em: <https://www.youtube.com/watch?v=xQQt-5myz0o>. Acesso em: 3 fev. 2020.

Um detalhe importante está no fato de que cada nó possui o par chave/valor. Entretanto, quando desenharmos árvores neste capítulo, colocaremos apenas o dado presente nas chaves. Isso porque o valor associado à chave é um dado útil apenas para quem usa o mapa, mas não é relevante para o estudo do funcionamento da árvore.

6.2.2 Informações básicas

A árvore possui o método `isVazio`, que retorna verdadeiro caso o nó raiz seja nulo. Uma opção equivalente seria retornar verdadeiro se o tamanho for 0. Outro método de informação é o método `getTamanho`, que retorna à quantidade de nós.

```
1  @Override
2  public int getTamanho() {
3      return tamanho;
4  }
5
6  @Override
7  public boolean isVazio() {
8      return raiz == null;
9  }
```

Outra informação interessante que poderia ser incluída é a altura da árvore. Porém, obtê-la pode ser complexo. Deixaremos essa implementação como um desafio para você.

6.2.3 Localizando itens

Como vimos no capítulo anterior, a principal função de um mapa é localizar itens de modo eficiente. Afinal, essa operação é necessária tanto para testar e localizar o valor associado à chave quanto para os processos de inserção e remoção de itens.

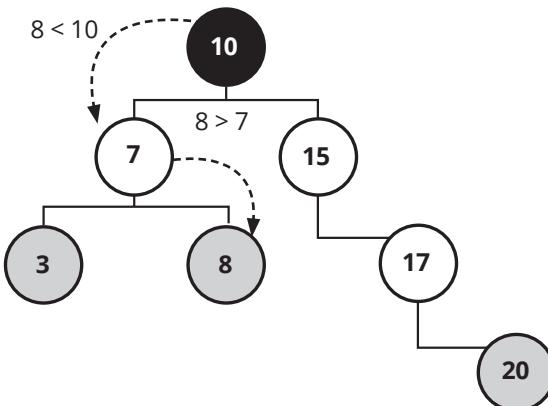
O processo de localização em uma árvore binária é descrito pelos seguintes passos (GOODRICH; TAMASSIA, 2013):

1. Inicie pelo nó raiz.
2. Se o nó atual for nulo, a chave não foi encontrada.
3. Caso contrário, teste se a chave procurada é igual à do nó atual.
Se for, retorne o valor associado, pois o elemento foi encontrado.
Se não for, navegue para o nó à esquerda, caso a chave desejada seja menor, ou para direita, caso seja maior. Repita o passo 2.

Observe que, em decorrência dos passos 1 e 2, se a raiz for nula, nenhuma chave jamais será encontrada, o que fará a função sempre retornar falso.

A figura a seguir ilustra a busca pela chave 8.

 Figura 4
Busca na árvore binária



Fonte: Elaborada pelo autor.

Para que a função possa ser útil tanto durante a busca quanto durante a inserção e remoção, precisaremos retornar outras informações adicionais (LAFORE, 2005):

- **O nó onde o dado foi encontrado**, que será nulo, caso ele não seja encontrado.
- **O nó pai**, que será nulo, caso o valor seja encontrado na raiz. Se o valor não for encontrado, será o nó que deverá ser pai dele no processo de inserção. Por exemplo, caso buscássemos pelo valor 9, o nó pai seria o nó 8. Isso é útil no processo de inserção.
- **A última comparação realizada**. Ela terá valor zero, caso o nó seja encontrado; valor negativo, caso ele devesse estar à esquerda do pai; e positivo se tivesse que estar à direita.

Para que esses três valores possam ser retornados, iremos criar outra classe estática privada auxiliar, denominada `NosRetorno`. Ela também possuirá os métodos `isEsquerda`, para indicar se o nó encontrado está ao lado esquerdo do seu pai, e `isRaiz`, caso o nó encontrado seja a raiz da árvore. A classe é descrita a seguir.

Desafio

Implemente a função `getAltura` que retorna à altura da árvore. Lembre-se de que a altura da árvore é igual à profundidade do nó mais profundo. Por isso, será necessário percorrer a árvore para descobrir essa informação.

```

1  private static class NosRetorno<C, V> {
2      No<C, V> pai;
3      No<C, V> no;
4      int cmp;
5
6      public boolean isRaiz() {
7          return pai == null;
8      }
9
10     public boolean isEsquerda() {
11         return pai != null && pai.esquerda == no;
12     }
13 }
```

Agora, basta implementarmos a função acharNos, que localiza esses nós seguindo a lógica já descrita:

```

1  private NosRetorno<C, V> acharNos(C chave) {
2      var ret = new NosRetorno<C, V>();
3      ret.no = raiz; //1. Inicie pela raiz
4
5      while(ret.no != null) { //2. O nó atual é nulo?
6          //3. Compare a chave com do nó
7          ret.cmp = comparator.compare(chave, ret.no.par.getChave());
8          if (ret.cmp == 0) {
9              return ret; //Se for igual, encontrou.
10         }
11
12         //Se não for, o nó atual será considerado o novo pai
13         ret.pai = ret.no;
14         //O próximo nó é definido indo para esquerda ou direta
15
16         ret.no = ret.cmp < 0 ? ret.no.esquerda : ret.no.direita;
17     }
18     //Não encontrou.
19     return ret;
20 }
```

Note que esta é uma função privada, uma vez que ainda lida com a estrutura interna da árvore em si.

6.2.4 Acesso direto a valores

Similarmente ao que ocorre no mapa não ordenado, o acesso direto a valores será implementado utilizando o método `acharNos`. Isso vale tanto para o método `get` quanto para o `contem`, conforme mostra o código a seguir.

```
1  @Override
2  public V get(C chave) {
3      var nos = acharNos(chave);
4      return nos.no == null ? null : nos.no.par.getValor();
5  }
6
7  @Override
8  public boolean contem(C chave) {
9      return acharNos(chave).no != null;
10 }
```

Lembre-se de que os retornos dos métodos como `get`, `adicionar` e `remover` indicam qual era o valor associado à chave antes do método ser chamado. Porém, há duas situações em que esses métodos podem retornar nulo: na ausência de uma chave, ou se o valor nulo for inserido ao mapa, associado àquela chave – o que também é válido. Assim, o método `contem` continua sendo a única forma inequívoca de se testar pela ausência de uma chave no mapa.

6.2.5 Adição de elementos

Embora adicionar chaves duplicadas seja uma operação válida para as árvores binárias, ela não é para mapas, e isso inclui o mapa ordenado que estamos implementando (ORACLE, 2019a). Por isso, o método `adicionar` irá substituir o valor, caso sua chave seja encontrada.

O processo de adicionar é simples:

1. Caso a árvore esteja vazia, criamos o nó raiz e o adicionamos.
2. Caso contrário, buscamos o nó que corresponde à chave que desejamos adicionar.
3. Se for encontrado, retornamos o valor antigo e o substituímos.
4. Se não for encontrado, adicionamos o nó ao lado da última comparação realizada (esquerdo, se a comparação for negativa, e direito, se positiva).

Analise que, caso a adição ocorra nos passos 1 ou 4, é necessário aumentar em um o valor da variável tamanho e retornar nulo.

```
1  @Override
2  public V adicionar(C chave, V valor) {
3      //1. Sem nós? Adiciona a raiz
4      if (isVazio()) {
5          raiz = new No<C>(chave, valor);
6          tamanho = tamanho + 1;
7          return null;
8      }
9
10     //2. Busca pelo nó correspondente à chave
11     var nos = acharNos(chave);
12
13     //3. Encontrou? Só substitui o valor
14     if (nos.cmp == 0) return nos.no.par.setValor(valor);
15
16     //4. Não encontrou? Adiciona de acordo com a última comparação
17     if (nos.cmp < 0) nos.pai.esquerda = new No<C>(chave, valor);
18     else nos.pai.direita = new No<C>(chave, valor);
19
20     tamanho = tamanho + 1;
21     return null;
22 }
```

Lembre-se de que o método setValor da classe Par retorna o valor anteriormente associado. Além disso, você também deve se lembrar de que é perfeitamente possível associar o valor null a uma chave, mas não é possível que haja uma chave nula.

6.2.6 Removendo elementos

A exclusão de elementos exige bem mais trabalho. O processo inicia-se localizando o nó que precisa ser excluído, o que é feito por meio da função acharNos. Se o nó não for encontrado, não há o que excluir, e simplesmente retornamos null:

```
1 | public V remover(C chave) {  
2 |     var nos = acharNos(chave);  
3 |     if (nos.no == null) {  
4 |         return null;  
5 |     }  
6 | }
```

Observe que o nó encontrado pode estar em três condições diferentes:

1. Ele pode ser uma folha, sem filhos.
2. Ele pode ter um único filho, seja ao lado esquerdo, seja direito.
3. Ele pode ter dois filhos.

Cada caso exige um tratamento diferenciado, sendo o último o mais complexo. Analisemos cada um deles a seguir.

6.2.6.1 Exclusão de folhas

Caso o nó seja uma folha, podemos removê-lo diretamente de seu pai. Mas há um caso especial a ser considerado: esse nó pode ser a raiz. Nesse caso, basta removê-lo da árvore:

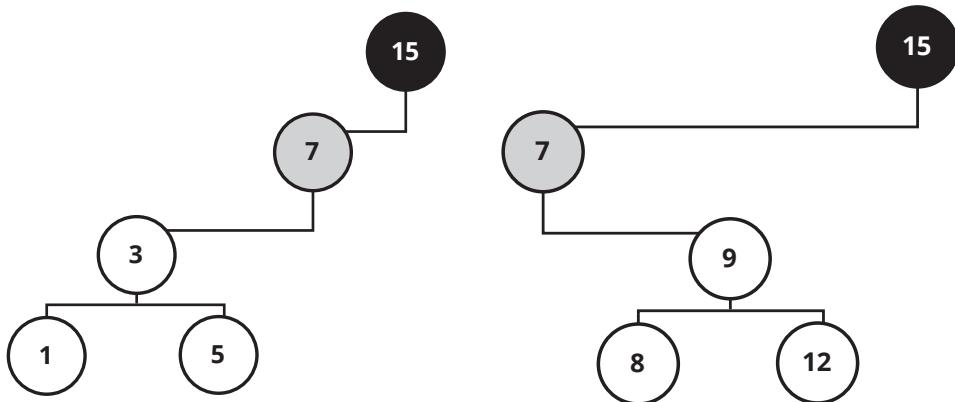
```
1 | var retorno = nos.no.par.getValor();  
2 |  
3 | //Não há filhos  
4 | if (nos.no.isFolha()) {  
5 |     if (nos.isRaiz()) {  
6 |         raiz = null;  
7 |     } else if (nos.isEsquerda()) {  
8 |         nos.pai.esquerda = null;  
9 |     } else {  
10 |         nos.pai.direita = null;  
11 |     }  
12 | }
```

6.2.6.2 Exclusão de nós com um filho

Observe as duas árvores a seguir, nas quais gostaríamos de excluir o nó de valor 7:



Figura 5
Exclusão do nó 7



Fonte: Elaborada pelo autor.

O nó 7 está à esquerda de seu pai. Portanto, todos os valores abaixo dele serão, necessariamente, inferiores ao valor presente no pai (15). Veja que essa condição é válida nas duas subárvore, não interessando se há nós ao lado direito ou esquerdo do nó 7, que está sendo excluído. O mesmo aconteceria caso o nó excluído estivesse à direita, mas, nesse caso, todos os valores seriam superiores a 15.

Portanto, a posição dos nós remanescentes se manterá inalterada em relação ao pai do nó que está sendo excluído. Isto é, no caso do exemplo, tanto o nó 3 da árvore da esquerda quanto o nó 9 da árvore da direita continuarão ao lado esquerdo do nó 15.

Consequentemente, o processo de exclusão de um só filho se resume a vincular esse filho, não interessando seu lado, ao pai do nó sendo excluído, na mesma posição em que esse nó sendo excluído está. O código é descrito a seguir.

```
1 //Somente um filho?
2 else if (nos.no.direita == null || nos.no.esquerda == null) {
3     //Obtém esse filho
4     var no = nos.no.direita == null ?
5         nos.no.esquerda : nos.no.direita;
6
7     //Substitui a raiz, ou associa ao pai, mantendo o lado
8     if (nos.isRaiz()) {
```

(Continua)

```

9      raiz = no;
10     } else if (nos.isEsquerda()) {
11         nos.pai.esquerda = no;
12     } else {
13         nos.pai.direita = no;
14     }
15 }
```

Note que também tivemos que considerar o caso especial da raiz. Como ela não possui pai, é ela que é diretamente substituída pelo filho em questão.

6.2.6.3 Exclusão de um nó com dois filhos

O processo de exclusão é um pouco mais complicado quando o nó possui dois filhos. A forma mais fácil de realizar uma exclusão é encontrar o sucessor imediato do nó sendo excluído (LAFORE, 2005). Esse valor pode ser encontrado ao navegar à direita tendo em vista o nó sendo excluído e procurar, em seguida, o elemento mais à esquerda.

Podemos criar a função `acharSucessor` para realizar essa tarefa:

```

1  private NosRetorno<C, V> acharSucessor(No<C, V> no) {
2      var ret = new NosRetorno<C, V>();
3      ret.pai = no;
4      ret.no = no.direita;
5      while (ret.no.esquerda != null) {
6          ret.pai = ret.no;
7          ret.no = ret.no.esquerda;
8      }
9      return ret;
10 }
```

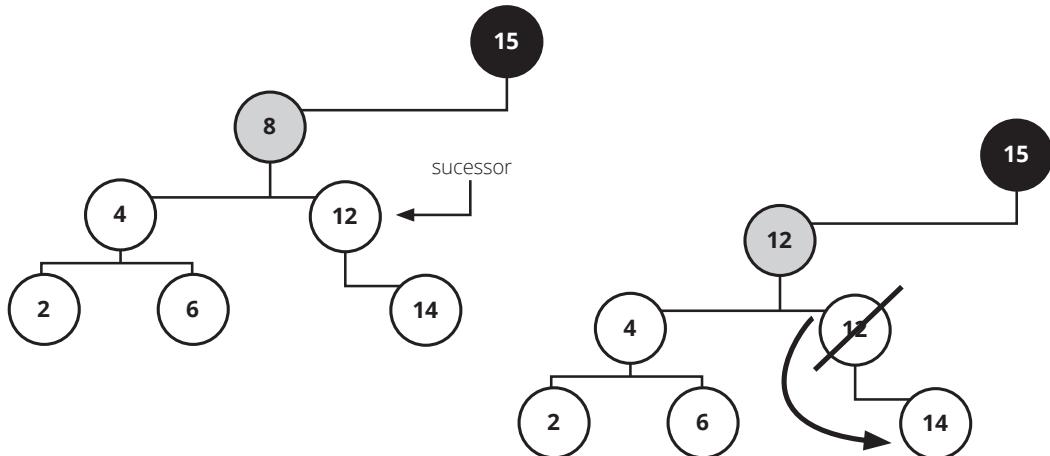
Em seguida, iremos copiar o par presente no sucessor para o interior do nó sendo excluído, fazendo com que os conteúdos dos dois nós se tornem iguais. Agora, basta excluir o nó em que o sucessor foi encontrado na árvore, que pode estar em dois casos distintos:

O sucessor está imediatamente à direita ao nó sendo excluído. Nesse caso, basta atualizar `no.direita` para o valor de `sucessor.direita`. Essa situação é ilustrada a seguir.



Figura 6

Exclusão do nó 8 com sucessor imediatamente à direita



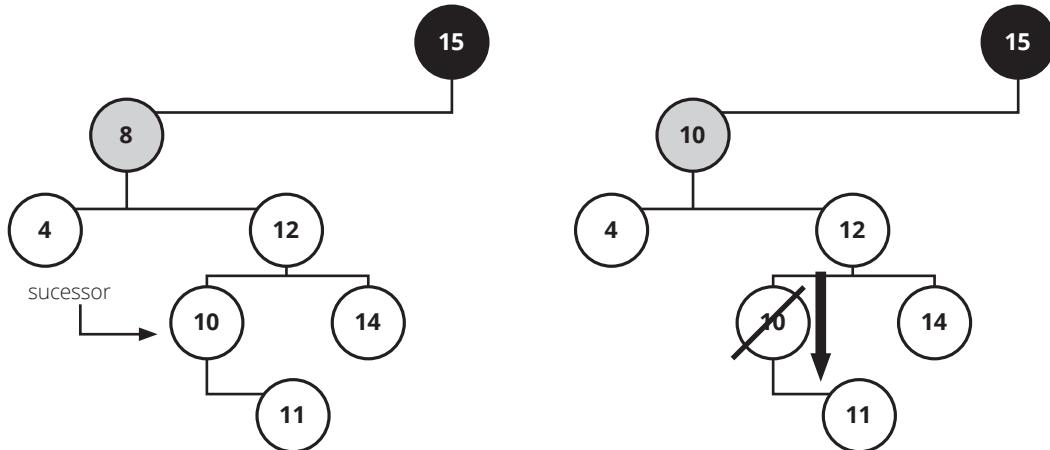
Fonte: Elaborada pelo autor.

O sucessor está em outra posição. Como o sucessor foi encontrado por meio de uma navegação à esquerda, devemos atualizar `sucessor.pai.esquerda` para o valor de `sucessor.direita`. Essa situação é ilustrada a seguir.



Figura 7

Exclusão do nó 8 com o sucessor 10 não estando imediatamente à direita



Fonte: Elaborada pelo autor.

Finalmente, em todos os casos, basta reduzir o tamanho da árvore em um e retornar o valor que estava associado ao nó excluído, armazenado na variável `retorno`.

Portanto, o código final para a exclusão do nó fica:

```
1 } else { //Dois filhos?  
2     var sucessor = acharSucessor(nos.no);  
3  
4     nos.no.par = sucessor.no.par;  
5     if (sucessor.no == nos.no.direita) {  
6         nos.no.direita = sucessor.no.direita;  
7     } else {  
8         sucessor.pai.esquerda = sucessor.no.direita;  
9     }  
10 }  
11  
12 tamanho = tamanho - 1;  
13 return retorno;  
14 }
```

Note que esse código funciona mesmo que o nó sendo excluído seja a raiz, ou caso o nó à direita do sucessor seja nulo.

6.2.7 Iteração

Árvores admitem três formas diferentes de iteração, denominadas *pré-ordem*, *em-ordem* e *pós-ordem* (GOODRICH; TAMASSIA, 2013). Implementaremos essas formas por meio de funções, similares à já conhecida função `forEach`.

Além dessas, também precisamos fornecer um `Iterator`. Ele iterará a árvore na forma *em-ordem*, porém precisamos considerar o caso da exclusão de itens.

6.2.7.1 Pré-ordem, em-ordem e pós-ordem

Na iteração *em-ordem*, seguimos três passos para cada nó:

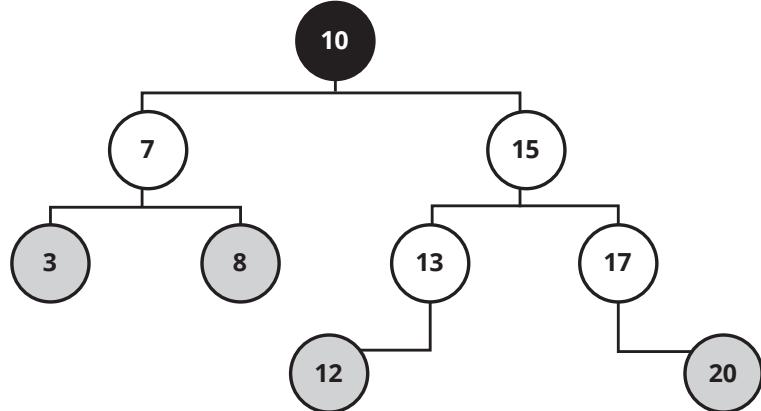
1. Caso haja um nó não visitado à esquerda, navegamos até ele.
2. Caso não haja, visitamos o nó atual.
3. Caso haja um nó à direita, navegamos até ele.

Voltamos ao nó pai até que a árvore esteja completamente visitada. E o que exatamente é “visitar um nó”? Significa que aquele é o momento de aquele nó ser utilizado. Ou seja, em uma iteração por meio da função `forEach`, é o momento em que o lambda, passado por nosso usuário para utilizar o nó, será chamado.

Observe que esse é um processo recursivo (MAIN; SAVITCH, 2005). Ou seja, ao navegarmos para um nó, seguiremos o passo a passo apresentado do início para aquele nó. Quando o passo a passo terminar para aquele nó, voltaremos ao nó anterior e continuaremos os itens no ponto onde foram abandonados.

Vejamos seu comportamento na árvore a seguir.

 **Figura 8**
Iteração *em-ordem* na árvore binária



1. O processo inicia-se na raiz.
2. A raiz possui um nó à esquerda. Nesse caso, desloca-se para o nó 7.
3. O nó 7 possui um nó à esquerda. Desloca-se para o nó 3.
4. O nó 3 não possui um nó à esquerda. **Visita o nó 3.**
5. O nó 3 não possui um nó à direita. Volta ao nó 7.
6. O nó à esquerda de 7 já foi visitado, por isso **visita o nó 7.**
7. O nó 7 possui um nó à direita. Navega até o nó 8.
8. O nó 8 não possui um nó à esquerda. **Visita o nó 8.**
9. O nó 8 não possui um nó à direita. Volta ao nó 7.
10. O nó 7 não possui nós não visitados à direita. Volta à raiz.
11. O nó à esquerda da raiz já está visitado. **Visita o nó 10 (raiz).**
12. A raiz possui um nó à direita. Navega até o nó 15.
13. O nó 15 possui um nó à esquerda. Navega até o nó 13.
14. O nó 13 possui um nó à esquerda. Navega até o nó 12.
15. O nó 12 não possui nós à esquerda. **Visita o nó 12.**
16. O nó 12 não possui nós à direita. Volta ao nó 13.
17. O nó à esquerda de 13 já foi visitado. **Visita o nó 13.**
18. O nó 13 não possui nós à direita. Volta ao nó 15.

(Continua)

19. O nó à esquerda de 15 já foi visitado. **Visita o nó 15.**
20. O nó 15 possui um nó à direita. Navega até o nó 17.
21. O nó 17 não possui nós à esquerda. **Visita o nó 17.**
22. O nó 17 possui um nó à direita. Navega até o nó 20.
23. O nó 20 não possui nós à esquerda. **Visita o nó 20.**
24. O nó 20 não possui nós à direita. Volta ao nó 17.
25. O nó 17 não possui mais nós a serem visitados. Volta ao nó 15.
26. O nó 15 não possui mais nós a serem visitados. Volta ao nó raiz.
27. A iteração termina.

Fonte: Elaborada pelo autor.

Note que os nós foram visitados na seguinte ordem: 3, 7, 8, 10, 12, 13, 15, 17 e 20; isto é, em ordem crescente. Esse é o objetivo da iteração *em-ordem*.

O que muda nas demais formas de iteração é a ordem em que o nó é visitado. A seguir, listamos as três formas.

1. Pré-ordem: visita o nó → navega para esquerda → navega para direita.
2. Em-ordem: navega para a esquerda → visita o nó → navega para direita.
3. Pós-ordem: navega para a esquerda → navega para direita → visita o nó.

Devido ao seu comportamento recursivo, a implementação dessas funções é bastante simples. Inicialmente, implementamos as funções que fazem a iteração com base em um nó específico, como a função `emOrdem`, a seguir.

```

1  private void emOrdem(No<C, V> no, Consumer<Par<C, V>> consumer) {
2      if (no == null) return;
3
4      emOrdem(no.esquerda, consumer);
5      consumer.accept(no.par);
6      emOrdem(no.direita, consumer);
7  }

```

Em seguida, fornecemos a versão pública, que faz sua chamada por meio do nó raiz:



Atividade 2

Descreva qual seria a ordem dos elementos da árvore da Figura 8, caso ela fosse impressa em pré-ordem e pós-ordem.

Atividade 3

Faça a implementação das funções `preOrdem` e `posOrdem`, de acordo com o que você aprendeu nesta seção.

```
1 | public void emOrdem(Consumer<Par<C, V>> consumer) {  
2 |     emOrdem(raiz, consumer);  
3 | }
```

A implementação das demais formas ficam muito similares, alterando-se apenas a posição da linha onde o `consumer` está, de acordo com o esquema desejado.

Isso nos permite imprimir todos os valores da árvore em uma determinada ordem com um comando simples, basta, para isso, passar a referência ao método `println`, como mostrado a seguir (ORACLE, 2019b):

```
arvore.emOrdem(System.out::println);
```

6.2.7.2 Implementando o Iterador

Seria extremamente complexo criar uma classe `ArvoreIterator` que percorra a árvore em ordem correta e ainda permita a exclusão dos elementos enquanto faz isso. Isso porque o último elemento percorrido pode estar muito distante do seu elemento anterior.

Porém, há um truque que nos permite facilmente implementar esse iterador. Podemos copiar todos os pares da árvore para uma lista estática com o mesmo tamanho da árvore. Como a informação de ordem dos nós foi copiada para essa estrutura auxiliar, a remoção poderá utilizar diretamente o método `remove()` da árvore.

Iniciamos o processo construindo um iterador para a árvore, que realiza essa tarefa com base no iterador da lista:

```
1 | private class ArvoreIterator implements Iterator<Par<C, V>>  
2 | {  
3 |     C ultimo = null;  
4 |     Iterator<Par<C, V>> it;  
5 |     public ArvoreIterator(Iterator<Par<C, V>> it) {  
6 |         this.it = it;  
7 |     }  
8 |  
9 |     @Override  
10 |     public boolean hasNext() {  
11 |         return it.hasNext();  
12 |     }  
13 | }
```

(Continua)

```

14     @Override
15     public Par<C, V> next() {
16         var par = it.next();
17         ultimo = par.getChave();
18         return par;
19     }
20
21     @Override
22     public void remove() {
23         remover(ultimo);
24     }
25 }
```

Veja que o método `next` armazena a chave do último elemento percorrido. Isso é feito para que a função `remover` da árvore possa ser chamada, caso a função `remove` do iterador seja chamada.

Agora, basta criamos a lista e preenchê-la, adicionando seus elementos, de acordo com a iteração *em-ordem* e, após isso, retornarmos o iterador:

```

1     @Override
2     public Iterator<Par<C, V>> iterator() {
3         var lista = new ListaEstatica<Par<C, V>>(getTamanho());
4         emOrdem(raiz, lista::adicionar);
5         return new ArvoreIterator(lista.iterator());
6     }
```

Lembre-se de que, conforme vimos no Capítulo 5, o mapa já possuiá métodos padrão para que a iteração possa ser feita também por meio das entradas, das chaves ou dos valores – todos com base na função `iterator` que acabamos de implementar.

6.3 Outros tipos de árvore



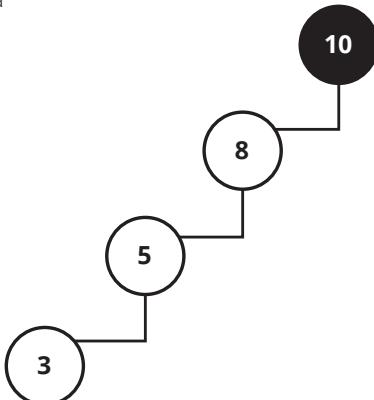
No início deste capítulo, vimos que árvores podem conter vários nós e que árvores binárias de busca são apenas um de seus tipos. Nesta seção, iremos explorar mais alguns tipos de árvore, entendendo que problema eles visam resolver.



6.3.1 Árvores-AVL

Um dos problemas com as árvores binárias de busca é que, muitas vezes, elas crescem apenas para um lado, dependendo da ordem que os elementos são inseridos. Por exemplo, considere a inserção dos elementos 10, 8, 5 e 3, nessa ordem. Isso geraria a seguinte árvore:

 Figura 9
Árvore desbalanceada



Fonte: Elaborada pelo autor.

Inserir elementos em ordem, seja ela crescente, seja decrescente, gera o seu pior caso: a árvore se comporta de maneira praticamente igual a uma lista encadeada (LAFORE, 2005). Isso ocorre porque a árvore perdeu seu **balanceamento**, isto é, porque não há uma boa distribuição de nós (GOODRICH; TAMASSIA, 2013).

Para resolver esse problema, os pesquisadores Adelson-Velsky e Landis criaram um processo de balanceamento automático. Nele, considera-se que cada nó possui um **fator de balanceamento** com base na altura de seus nós filhos. Esse fator é dado por (LAFORE, 2005):

$$\text{fatorBalanceamento}(n) = \text{altura}(\text{esquerda}(n)) - \text{altura}(\text{direita}(n))$$

Todo nó em que o fator de balanceamento não possui os valores -1, 0 ou 1 deve ser rebalanceado. Isso é feito realizando uma de quatro rotações possíveis: esquerda, direita, esquerda-direita e direita-esquerda.

6.3.1.1 Rotações para a esquerda ou direita

As rotações para esquerda ou direita são análogas. Ocorrem quando os nós são adicionados duas vezes para o mesmo lado. Nesse caso,

Site

O site *AVL Tree* apresenta uma forma interativa de visualizar as árvores AVL e o processo de平衡amento em ação. Procure inserir nós em ordem e visualizar os processos descritos neste capítulo.

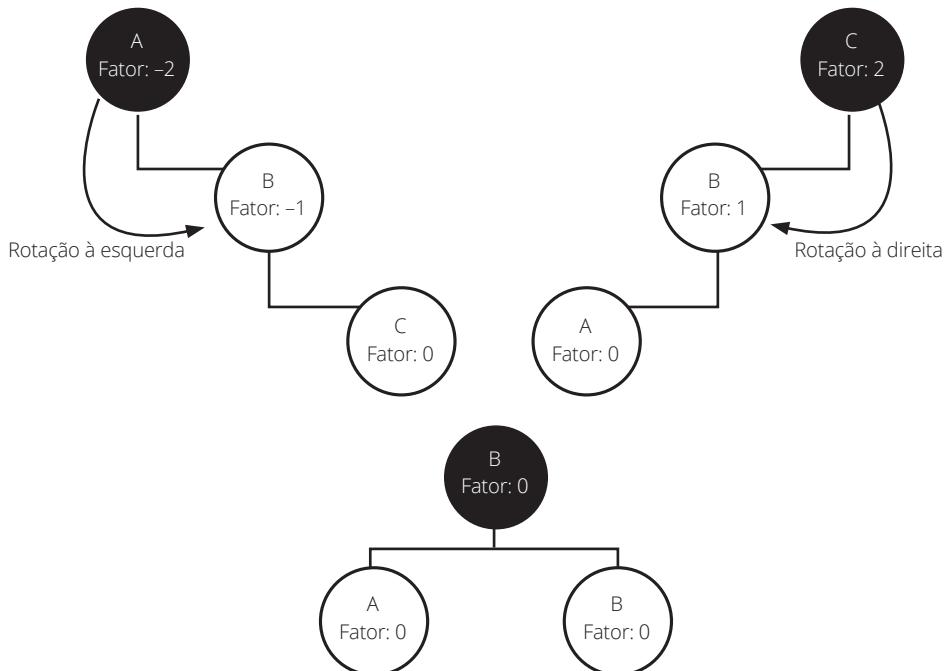
Disponível em: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>. Acesso em: 3 fev. 2020.

o nó central deverá se tornar o novo pai do conjunto. Esse problema é ilustrado na figura a seguir. Observe que, em ambos os casos, o resultado será igual após as rotações.



Figura 10

Rotação para a direita e para a esquerda e resultado após rotação



Fonte: Elaborada pelo autor.

Note que, ao final do processo, o fator de carga dos três nós é 0, o que garante máxima eficiência na nova árvore formada.

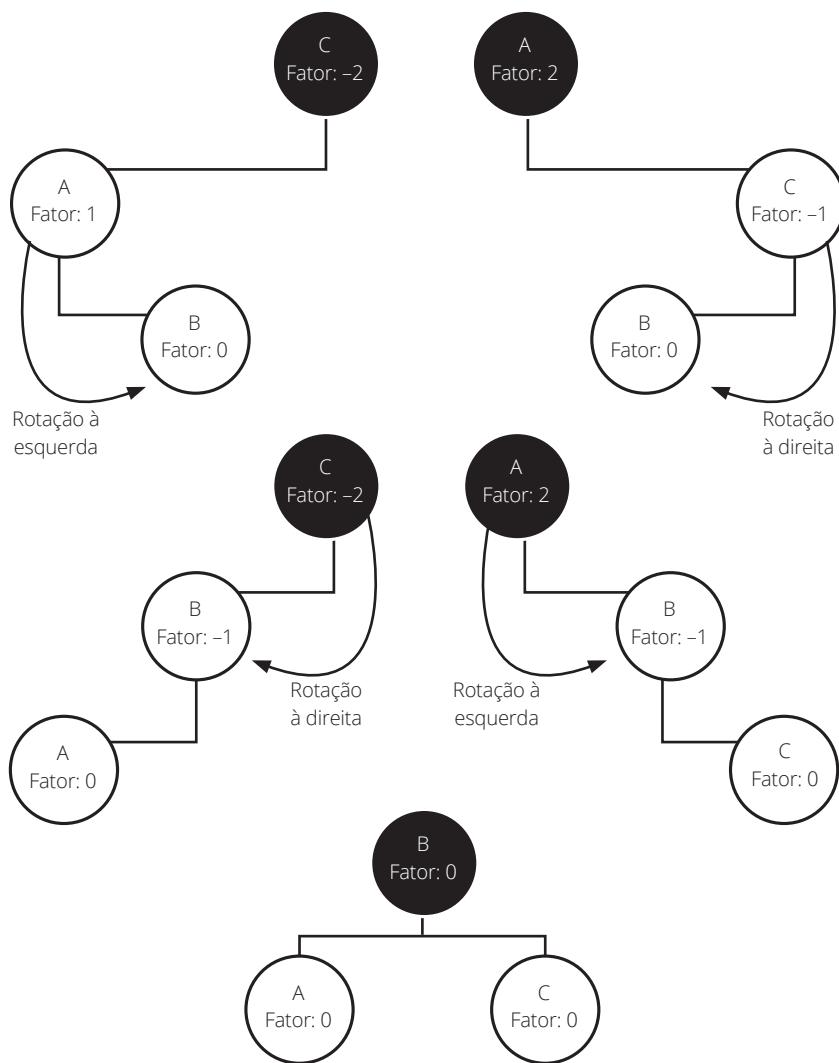
6.3.1.2 Rotações esquerda-direita e direita-esquerda

Às vezes, o desbalanceamento ocorre quando os nós são inseridos em zigue-zague – ou seja, primeiro à esquerda, seguido de um nó abaixo deste à direita; ou primeiro à direita, com um nó abaixo deste à esquerda. Nesse caso, são necessárias duas rotações de modo a reabalancear a árvore (GOODRICH; TAMASSIA, 2013). Analise as operações na figura a seguir.



Figura 11

Rotações esquerda-direita e direita-esquerda



Fonte: Elaborada pelo autor.

Observe que, após as rotações, a árvore encontra-se de novo perfeitamente balanceada. Esse processo contínuo de rebalanceamento aumenta um pouco o custo de inserção, mas melhora significativamente a performance das buscas, que se mantém em tempo logarítmico – uma das grandes vantagens das árvores AVL. Mesmo com esse aumento, o custo ainda é considerado relativamente baixo (GOODRICH; TAMASSIA, 2013).

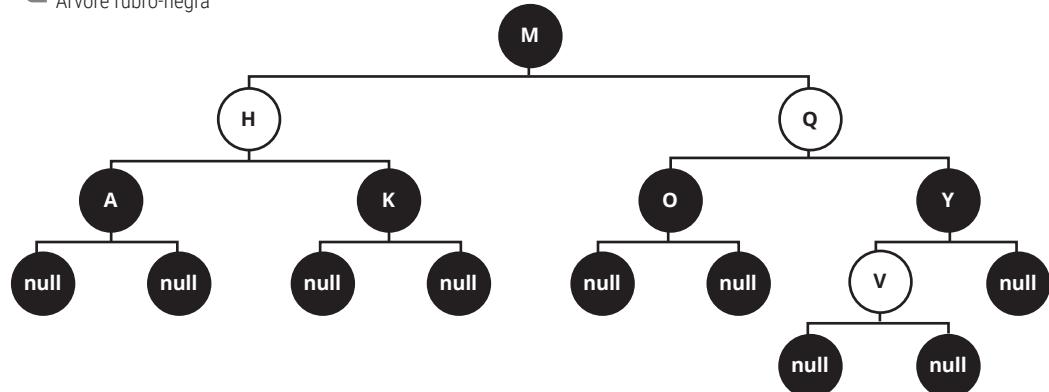
6.3.2 Árvores rubro-negras

Árvores rubro-negras ou árvores vermelho e preto são outra estratégia para tentar manter o balanceamento de árvores binárias. Nelas, associam-se as cores vermelha ou preta a cada nó, tentando respeitar quatro regras (LAFORE, 2005):

1. A raiz da árvore é sempre preta.
2. Os valores nulos ao lado esquerdo e direito das folhas são considerados pretos.
3. Todos os filhos de um nó vermelho são pretos.
4. Todo caminho de um nó até qualquer uma de suas folhas terá o mesmo número de nós pretos.

A seguir, podemos ver um exemplo desse tipo de árvore.

 Figura 12
Árvore rubro-negra



Os nós vermelhos foram desenhados em branco.

Fonte: Elaborada pelo autor.

Não entraremos em detalhes sobre as operações necessárias para garantir essa condição. Porém, é interessante saber que as árvores rubro-negras têm performance superior à AVL no pior caso, embora apresente desempenho ligeiramente inferior no caso mais comum (LAFORE, 2005).

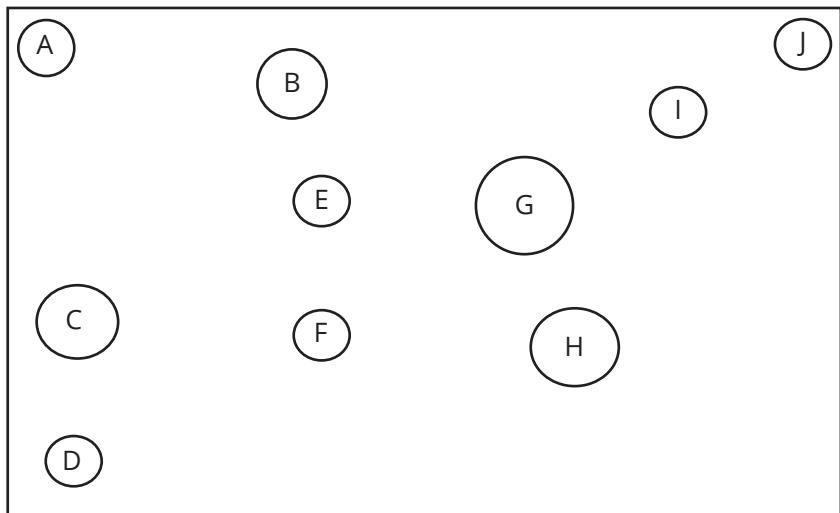
6.3.3 Árvores m-ways

No início deste capítulo, vimos que a limitação de ter apenas dois nós é exclusiva das árvores binárias. Já árvores m-ways têm como objetivo criar uma árvore de busca de modo que utilizemos até m nós (LAFORE, 2005).

Um exemplo de m-way bastante utilizado em aplicações gráficas, como jogos ou mapas, é a **quadtree** (4-way). Considere o problema de testar se o mouse foi clicado em um objeto da tela. O conjunto de objetos possível pode ser bastante grande, como ilustra a figura a seguir.

 Figura 13

Objetos clicáveis na tela



Fonte: Elaborada pelo autor.

Cada forma exigirá um cálculo matemático para saber se a colisão com o mouse ocorreu. Com formas mais complexas que os círculos do exemplo (as quais ocorrem em problemas reais), esse cálculo pode ser bastante custoso (DUNN; PARBERRY, 2011). Assim, com muitas formas, esse processo se tornaria rapidamente ineficiente, caso tenhamos que testar se a colisão ocorreu objeto a objeto.

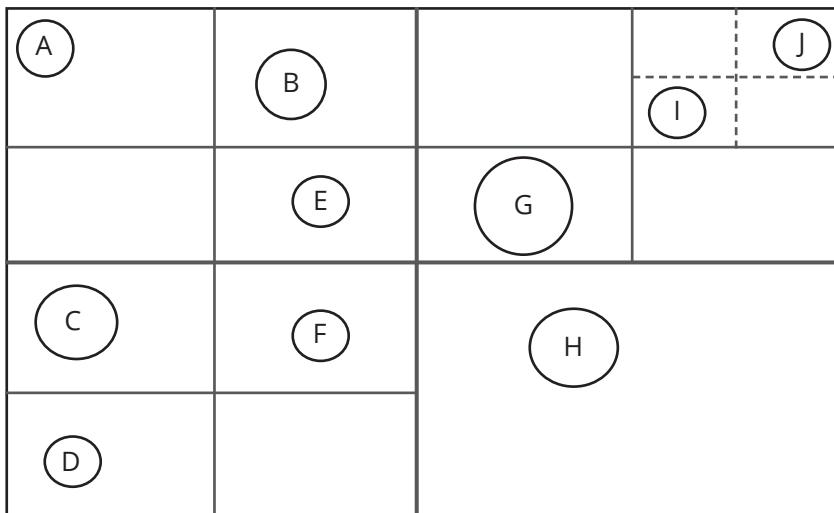
Sabemos que, se o clique do mouse ocorreu do lado direito da tela, ele jamais atingirá um objeto do lado esquerdo, e vice-versa. Com base nessa observação, uma solução possível seria subdividir a tela em quatro espaços e testar primeiro a colisão com esses espaços. Como os espaços são retângulos, seu cálculo de colisão terá um custo computacional baixo, se comparado a formas mais complexas (DUNN; PARBERRY, 2011).

Cada espaço terá subordinado a si os objetos que se encontram dentro dele. Isso porque temos certeza de que, se o espaço onde o objeto está não foi clicado, o objeto que está dentro dele também não foi.

Note que, se as quatro subdivisões ainda representarem uma área muito grande, poderíamos repetir a estratégia em seu interior, divi-

dindo cada espaço novamente em quatro subdivisões subordinadas. Também, repare que nunca precisamos subdividir áreas vazias ou contendo apenas um objeto em seu interior, independentemente de seu tamanho, conforme ilustrado a seguir.

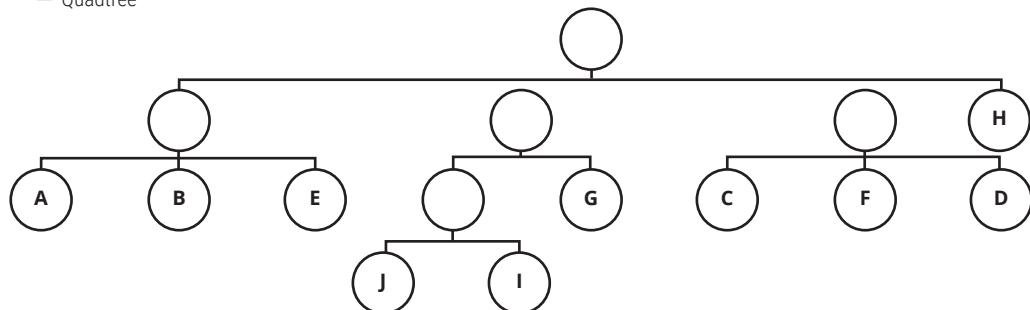
 Figura 14
Espaço subdividido



Fonte: Elaborada pelo autor.

Essa organização representa uma estrutura de árvore 4-way, em que cada subdivisão é um nó. Espaços vazios não precisam ser inseridos na árvore. Isso resulta na seguinte quadtree:

 Figura 15
Quadtree

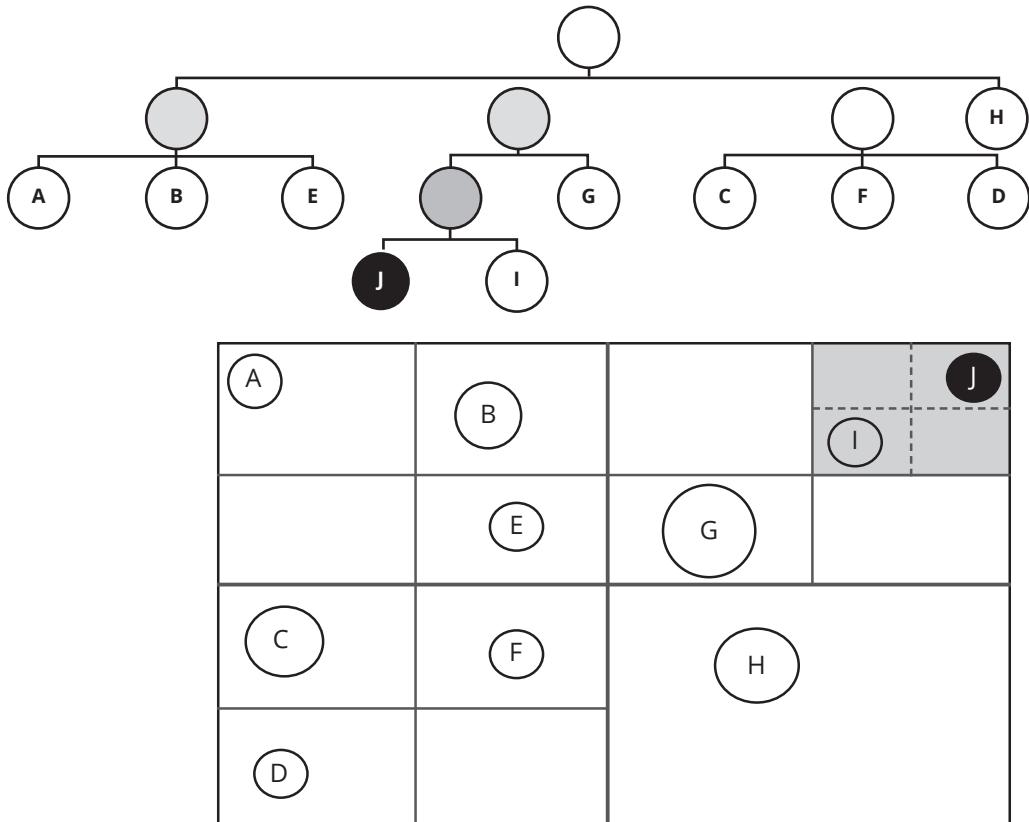


Fonte: Elaborada pelo autor.

Dessa forma, para testar se o objeto J foi clicado, seriam feitos apenas quatro testes de colisão, conforme ilustrado a seguir. Não seria necessário fazer testes contra o nó raiz, que representa todo o espaço clicável da tela:

 Figura 16

Testes de colisão para encontrar o objeto J

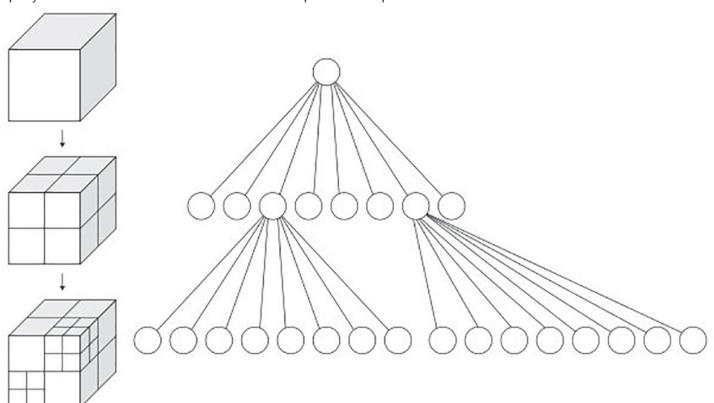


Fonte: Elaborada pelo autor.

Uma variação das quadtrees para espaços tridimensionais são as **octrees**. A diferença está apenas no fato de que elas também dividem o espaço no eixo z:

 Figura 17

Espaço 3D subdividido em níveis de 8 para compor a octree



WhiteTimberwolf/Wikimedia Commons

Embora não seja tão intuitivo, é possível usar m-ways para informação unidimensional, como valores numéricos, dividindo os dados entre diferentes faixas de valores.



CONSIDERAÇÕES FINAIS

As estruturas de dados escolhidas em nosso estudo não foram acidentais. Elas são as estruturas que nos permitem entender a Java Collections Framework. A biblioteca de coleções implementa várias melhorias nos algoritmos descritos aqui, além de versões mais otimizadas ou enxutas dos códigos apresentados, mas nosso estudo já nos dá uma boa visão dos custos computacionais envolvidos em cada coleção.

Devido a sua extensão e complexidade, esse material não contempla o estudo de uma estrutura de dados chamada *grafo*, o qual permite que modelemos nós relacionados praticamente de qualquer forma. É com grafos que podemos mapear as ruas entre dois pontos em um mapa e utilizar algoritmos para achar o menor caminho entre eles. Considere estudá-lo por conta própria, caso tenha interesse em se aprofundar no estudo das estruturas de dados. As árvores, vistas neste capítulo, são consideradas um tipo restrito de grafo.

Esperamos que você tenha aproveitado nossa abordagem prática sobre o assunto. Intencionalmente, deixamos de lado as demonstrações matemáticas ou explicações por meio de cálculos. Você pode acabar se deparando com elas ao procurar por esse tema na internet, pois são importantes para matemáticos e pesquisadores interessados na teoria de dados – mas pouco úteis para a maior parte dos profissionais da computação do dia a dia.

Procure dedicar um tempo para implementar cada uma das estruturas do livro e entendê-las. Ao longo de sua carreira, você ficará impressionado pela quantidade de vezes que esse assunto será útil – mesmo que seja simplesmente para entender melhor as classes de estruturas de bibliotecas já implementadas para você. Afinal, fazer boas escolhas e entender bem o impacto dos algoritmos é importante em qualquer área que você queira atuar.



REFERÊNCIAS

- DUNN, F.; PARBERRY, I. *3D Math Primer for Graphics and Game Development*. 2. ed. Boca Raton, FL: CRC Press, 2011.
- GOODRICH, M. T.; TAMASSIA, R. *Estruturas de Dados & Algoritmos em Java*. 5. ed. Porto Alegre: Bookman, 2013.
- LAFORE, R. *Estruturas de dados & algoritmos em Java*. 2. ed. Rio de Janeiro: Ciência Moderna, 2005.
- MAIN, M.; SAVITCH, W. *Data structures & Other objects using C++*. 3. ed. Boston, MS: Pearson, 2005.

ORACLE. *Interface Map<K,V>*. 2019a. Disponível em: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/Map.html>. Acesso em: 24 jan. 2020.

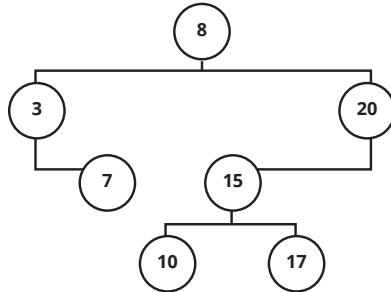
ORACLE. Method References. *The Java Tutorials*. 2019b. Disponível em: <https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>. Acesso em: 24 jan. 2020.



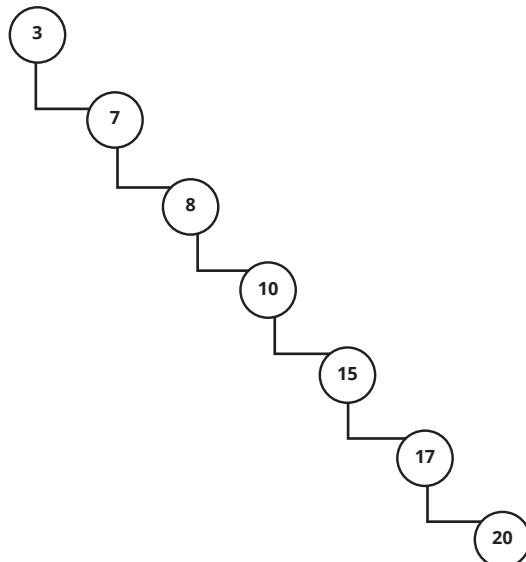
GABARITO

1.

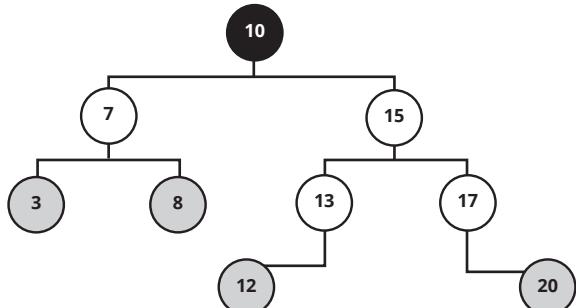
- a. 8, 3, 7, 20, 15, 17, 10



- b. 3, 7, 8, 10, 15, 17, 20



2.



Pré-ordem: 10, 7, 3, 8, 15, 13, 12, 17, 20.

Visita o nó → navega para esquerda → navega para direita.

1. Inicia na raiz (nó 10). Visita a raiz.
2. Navega para a esquerda, indo para o nó 7. **Visita o nó 7.**
3. Navega para a esquerda, indo para o nó 3. **Visita o nó 3.**
4. Não há nós para navegar à esquerda de 3.
5. Não há nós para navegar à direita de 3. Volta ao nó 7.
6. Navega para a direita (nó 8). **Visita o nó 8.**
7. Não há nós para navegar à esquerda de 8.
8. Não há nós para navegar à direita de 8. Volta ao nó 7.
9. Não há mais ações no 7. Volta ao nó raiz 10.
10. Navega para a direita (nó 15). **Visita o nó 15.**
11. Navega para a esquerda, indo para o nó 13. **Visita o nó 13.**
12. Navega para a esquerda, indo ao nó 12. **Visita o nó 12.**
13. Não há nós para navegar à esquerda de 12.
14. Não há nós para navegar à direita de 12. Volta ao nó 13.
15. Não há nós para navegar à direita de 13. Volta ao nó 15.
16. Navega a direita de 15, para o nó 17. **Visita o nó 17.**
17. Não há nós para navegar à esquerda de 17.
18. Navega para a direita de 17, para o nó 20. **Visita o nó 20.**
19. Não há nós para navegar à esquerda de 20.
20. Não há nós para navegar à direita de 20.
21. Volta ao nó 17. Volta ao nó 15. Volta ao nó 10.
22. A iteração termina.

Pós-ordem: 3, 8, 7, 12, 13, 20, 17, 15, 10.

Navega para a esquerda → navega para direita → visita o nó

1. Inicia na raiz. Navega para a esquerda, para o nó 7.
2. Navega para a esquerda do nó 7, indo ao nó 3.
3. Não há nós à esquerda do nó 3.
4. Não há nós à direita do nó 3.
5. **Visita o nó 3.** Volta ao nó 7.
6. Navega para a direita do nó 7, indo ao nó 8.
7. Não há nós à esquerda do nó 8.
8. Não há nós à direita do nó 8.
9. **Visita o nó 8.** Volta ao nó 7.
10. **Visita o nó 7.** Volta ao nó 10.
11. Navega para a direita do nó 10, indo ao nó 15.
12. Navega para a esquerda do nó 15, indo ao nó 13.
13. Navega para a esquerda do nó 13, indo ao nó 12.
14. Não há nós à esquerda do nó 12.
15. Não há nós à direita do nó 12.
16. **Visita o nó 12.** Volta ao nó 13.
17. Não há nós à direita de 13.
18. **Visita o nó 13.** Volta ao nó 15.

19. Navega para o nó à direita de 15, indo ao nó 17.
20. Não há nós à esquerda de 17.
21. Navega para o nó à direita do 17, indo ao nó 20.
22. Não há nós à esquerda de 20.
23. Não há nós à direita de 20.
- 24. Visita o nó 20.** Volta ao nó 17.
- 25. Visita o nó 17.** Volta ao nó 15.
- 26. Visita o nó 15.** Volta ao nó 10.
- 27. Visita o nó 10.**
28. A iteração termina, pois é a raiz.

3.

```

1  private void preOrdem(No<C, V> no, Consumer<Par<C, V>> consumer) {
2      if (no == null) return;
3
4      consumer.accept(no.par);
5      preOrdem(no.esquerda, consumer);
6      preOrdem(no.direita, consumer);
7  }
8
9  private void posOrdem(No<C, V> no, Consumer<Par<C, V>> consumer) {
10     if (no == null) return;
11
12     posOrdem(no.esquerda, consumer);
13     posOrdem(no.direita, consumer);
14     consumer.accept(no.par);
15 }
16
17 public void preOrdem(Consumer<Par<C, V>> consumer) {
18     preOrdem(raiz, consumer);
19 }
20
21 public void posOrdem(Consumer<Par<C, V>> consumer) {
22     posOrdem(raiz, consumer);
23 }
```


Informática vem do francês *information automatique* (informação automática). No passado, também foi chamada de processamento de dados. Não é mera coincidência que o nome da área de informática sempre se refira a dados e à informação.

Hoje os computadores são responsáveis por processar um volume enorme de dados com extrema rapidez. Entretanto, por mais rápidos que se tornem, é importante que o programador utilize bons algoritmos, capazes de gerenciar todos os recursos com a máxima eficiência.

Ao iniciar um projeto, seja o sistema de um grande banco ou um aplicativo para o celular, várias questões surgem, como: qual é o volume de dados que será processado? O acesso a esses dados ocorre em sequência? Há alguma forma de ordenar esses dados? As informações são independentes ou estão relacionadas? Com este livro, você conseguirá responder a essas perguntas ao estudar sobre estruturas de dados, suas características técnicas e formas de implementação.

Fundação Biblioteca Nacional
ISBN 978-85-387-6573-8



9 788538 765738



59130

