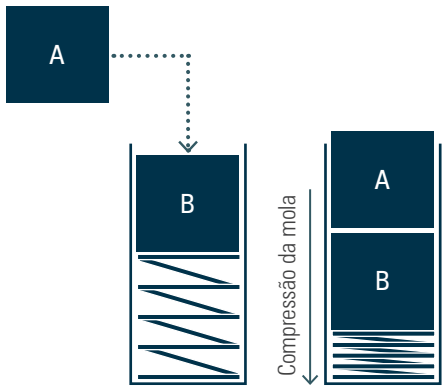
Capítulo 7

Pilha

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

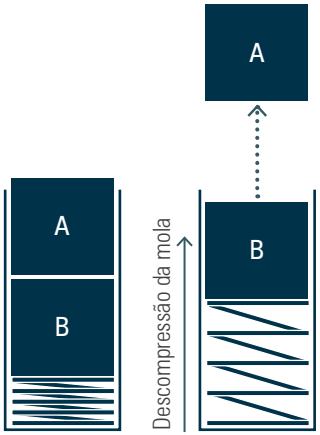
Alguns exemplos de aplicações de pilha no cotidiano: uma pilha de pratos para lavar; a organização das entregas dentro dos caminhões das empresas de logística; os sistemas de solução de equações matemáticas; os dispensadores de objetos, como nas máquinas de venda automática; entre muitos outros cenários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

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



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

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



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

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

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

- ***topo* (*top*, em inglês):** consulta qual o elemento está na posição do topo da pilha. Esse método não deve remover o elemento, ele é apenas informativo.

- ***empilha* (*push*, em inglês):** insere um elemento no topo da pilha. Esse método deve receber como parâmetro o elemento que é inserido na pilha.

- ***desempilha* (*pop*, em inglês):** remove e retorna o elemento que está no topo da pilha. Uma regra importante nesse método é que nenhum elemento pode ser removido de uma pilha vazia, portanto uma mensagem de erro deve retornar caso isso ocorra (GOODRICH; TAMASSIA, 2013).

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

1 Implementando a pilha com o uso de vetores

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.1 Ampliando a capacidade da pilha

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

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

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

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

```
11.    public int topo() {
12.           return this.dados.get(this.dados.size() -
1);
13.    }
14.
15.    public int tamanho() {
16.           return this.dados.size();
17.    }
18.
19.    public void empilha(int numero) {
20.           this.dados.add(numero);
21.    }
22.
23.    public int desempilha() {
24.           if(this.ehVazia()) {
25.                  throw new RuntimeException("A pilha
está vazia.");
26.           }
27.           return this.dados.remove(this.dados.size() -
1);
28.    }
29.
30.    @Override
31.    public String toString() {
32.           return this.dados.toString();
33.    }
34. }
```

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

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

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

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

2 Utilizando uma lista ligada para empilhar objetos

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

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

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

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

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

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

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

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

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

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

PARA SABER MAIS

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

3 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

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

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

3. Crie um sistema de logística que deverá gerar uma lista de entregas para um caminhão, no qual o primeiro pacote alocado no caminhão deverá ser o último a chegar ao destino. Considere também que o último destino é o local mais próximo da transportadora.

4. Crie uma simulação do sistema de desfazer uma alteração no editor de texto, ou seja, o comando gerado pela ação de pressionar as teclas Control e Z.

Considerações finais

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

Referências

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

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

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

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

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