Capítulo 6

Filas

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

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

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

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

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

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

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

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

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

1 Implementando a fila com vetores

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

PARA SABER MAIS

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

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

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

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

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

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

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

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

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

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

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

```
1. public class App {
2.     public static void main(String[] args) {
3.
4.            Fila f = new Fila(3);
5.
6.            System.out.printf("Fila atual: %s.\n",
f.toString());
7.
8.            try {
9.                   f.enfileira(29);
10.                  System.out.printf("Fila %s <= %s
enfileirado.\n", f.toString(), 29);
11.                  f.enfileira(10);
12.                  System.out.printf("Fila %s <= %s
enfileirado.\n", f.toString(), 10);
```

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

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

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

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

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

1.1 Resolvendo o problema de capacidade máxima

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

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

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

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

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

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

1.2 Analisando o desempenho das operações

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

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

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

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

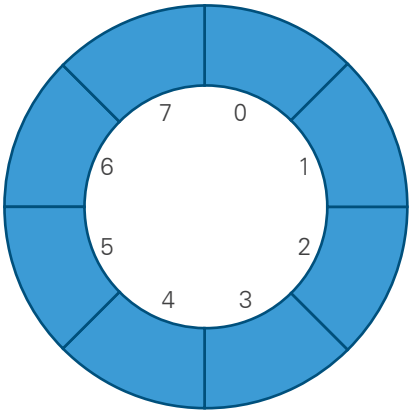
MÉTODO	COMPLEXIDADE
primeiro	O(1)
ehVazia	O(1)
tamanho	O(1)
enfileira	O(1)
desenfileira	O(*n*)

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

2 Andando em círculo

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

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



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

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

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

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

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

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

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

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

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

3 Ligando os elementos

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

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

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

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

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

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

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

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

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

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

MÉTODO	COMPLEXIDADE
primeiro	O(1)
ehVazia	O(1)
tamanho	O(1)
enfileira	O(1)
desenfileira	O(1)

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

PARA SABER MAIS

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

4 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Utilizando o método System.currentTimeMillis(), disponível no Java, compare o tempo de processamento dos métodos *desenfileira* apresentados ao longo do capítulo.

2. Crie duas filas de atendimento bancário, uma com atendimento normal e outra de atendimento prioritário. A fila de atendimento prioritário deve ser duas vezes mais rápida que a de atendimento normal. Teste sua implementação.

3. Pesquise a aplicação de fila em escalonadores *round-robin*.

4. Implemente um escalonador *round-robin* para um sistema de logística em que é necessário carregar caminhões para entrega.

Considerações finais

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

Referências

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

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

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

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