ESTRUTURA DE DADOS

UNIDADE 3 – LISTA, FILA E PILHA

Katia Cristina Lage dos Santos

Introdução

Um tipo abstrato de dados é formado por um mecanismo de armazenamento e um conjunto de operações quanto a esses dados. Uma analogia interessante é uma sala de aula, em que os elementos que a formam são as mesas e as cadeiras, dispostas sequencialmente. Algumas operações que podem ser aplicadas sobre esses dados seriam mover a cadeira, escrever sobre a mesa, entre outras. No entanto, e se quiséssemos criar um mecanismo mais elaborado para definir a inclusão ou remoção de novas mesas e cadeiras na sala?

Os aspectos relacionados à inserção e retirada de elementos em uma estrutura de dados são operações muito recorrentes. De acordo com a ordem em que esses elementos são inseridos ou removidos, temos três importantes estruturas de dados. Será, contudo, que essas estruturas podem ser implementadas de forma diferente no paradigma estruturado e no paradigma orientado a objetos? Qual é a complexidade disso para entendermos as estruturas?

Nesta unidade, veremos os conceitos de fila, pilha e lista, que, assim como diversos outros na área de Ciência da Computação, visam imitar os conceitos do mundo real. Essa abstração será importante para compreendermos a definição e, com a prática, aplicarmos a estrutura correta para cada situação.

Além disso, aprenderemos que, independentemente do paradigma de programação adotado, o conceito das estruturas de dados será o mesmo. Projetos de expressão nacional, como o Sistema Público de Escrituração Digital, é um exemplo, pois utilizam o conceito de filas para realizar o processamento das notas fiscais.

Vamos, então, iniciar nossos estudos. Acompanhe!

3.1 Lista

O termo “lista” é comumente empregado no dia a dia. Temos listas de tarefas, de supermercados ou até de presentes de casamento. Contudo, do que, de fato, trata-se uma lista?

De forma geral, uma lista se refere a um conjunto de elementos dispostos de forma sequencial, podendo ou não ter repetição do mesmo item. De maneira mais simples, poderíamos representar uma lista como uma sequência de itens:

- Arroz
- Feijão
- Sabão em pó
- Detergente
- Leite
- Café

Só que as listas também podem ser mais elaboradas. Se, por exemplo, pensarmos em agrupar os itens anteriores em categorias, poderíamos ter uma representação um pouco diferente, como:

1. Alimentos
- arroz;
- feijão;
- leite;
- café.
2. Limpeza:
- sabão em pó;
- detergente.

Nas duas representações de listas, é possível perceber que, quando vamos fazer a leitura, inserir ou retirar um elemento, podemos realizar a ação de forma arbitrária. Por exemplo, imagine a situação de ter uma dessas listas para fazer compras no supermercado: poderíamos ler item a item, na ordem, e ir marcando aqueles que já estão no carrinho de compras. Assim, essas características são mimetizadas na estrutura de dados de mesmo nome.
Na linguagem Java, a forma mais simples para armazenar os elementos de uma lista é com o uso de vetores (*arrays*), também conhecidas como listas implementadas com *arrays* (PUGA; RISSETTI, 2010). Entretanto, como a inserção ou retirada de elementos pode ocorrer em qualquer posição do vetor, existem maneiras diferentes de implementar essa estrutura de dados.

VOCÊ QUER LER?

Podemos conseguir diversos materiais adicionais para o estudo e entendimento das estruturas de dados na internet. Se aprofundar a respeito do assunto é bastante interessante, pois é possível ter acesso à implementação das estruturas em diferentes linguagens e com abordagens distintas. Nesse sentido, uma referência para leitura é a apostila elaborada pelo professor Ivan Ricarte, da Unicamp, intitulada "Estrutura de Dados", disponível no *link*: http://calhau.dca.fee.unicamp.br/wiki/images/0/01/EstruturasDados.pdf.

No item a seguir, veremos os tipos de listas e como a implementação delas pode ser feita com o uso de classes.

3.1.1 Tipos de listas

Pela característica estática dos vetores, adicionar um elemento na primeira posição, por exemplo, consome muito tempo, pois precisamos deslocar todos os outros elementos uma posição para frente. A *performance* dessa operação degrada conforme a quantidade de elementos do nosso vetor cresce, ou seja, ela consome tempo linear em relação ao número de elementos. Analogamente, remover um elemento da primeira posição implica em deslocar todos os outros elementos que estão na frente para trás.
Nessas situações, devemos fazer uso das classes e dos objetos para conseguirmos adicionar ou remover um elemento de alguma posição da lista, com um custo computacional menor que a manipulação de vetores. Nesse sentido, surge o conceito das **listas encadeadas**. A figura a seguir representa este modelo.

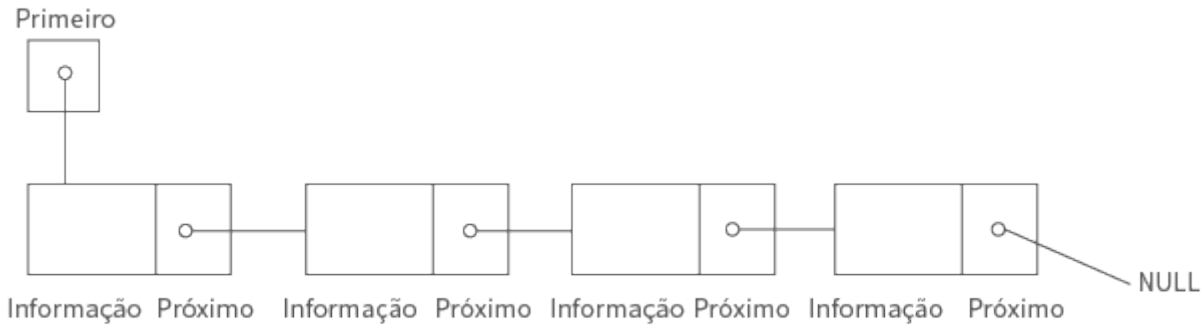

Figura 1 - Representação de uma lista encadeada com quatro elementos
Fonte: Elaborada pela autora, 2019.

Uma lista encadeada, então, corresponde a uma estrutura iniciada pela definição de um elemento que "aponta" para o início da lista. Na figura anterior, o elemento "primeiro" está indicando qual é o primeiro item da lista.
Cada elemento é formado por duas partes: uma informação e uma referência (próximo) para o próximo elemento da lista. Isto ocorre para todos os elementos, até que seja alcançado o último. Este, de forma diferenciada, tem um marcador para indicar o fim da lista. Geralmente, o marcador utilizado para o fim da lista é o valor *null*.

VOCÊ O CONHECE?

Donald Ervin Knuth é um cientista da computação de renome e professor emérito da Universidade de Stanford. Ele é comumente referenciado pelas análises cuidadosas de algoritmos clássicos da Ciência da Computação. Autor do livro "A arte da Ciência da Computação", uma das principais referências da área, Knuth também é o criador do sistema tipográfico TEX, bastante utilizado para a construção de artigos científicos (ROBERTS, 2018). Vale conhecermos mais sobre ele!

Para a representação em Java da estrutura de dados, faremos o uso de classes. Na figura a seguir, é declarada a classe "lista", que contém uma referência para o primeiro item. Cada um dos elementos é apresentado pela classe "ItemLista", que contém um atributo do tipo *string*, representando a informação e uma referência (*prox*) para o próximo item da lista.

```
1  class Lista{
2      private ItemLista primeiroItem;
3  }
4
5  class ItemLista{
6          private String info;
7          private ItemLista prox;
8  }
```

Figura 2 - Implementação da lista simplesmente encadeada em Java
Fonte: Elaborada pela autora, 2019.

A lista apresentada é chamada de "simplesmente encadeada", pois, para cada célula ou item, há a indicação apenas do próximo elemento da lista.
No entanto, existe outro tipo de lista em que há, também, um indicador para o elemento anterior. Como há uma indicação dupla, a lista é chamada de "duplamente encadeada". Veja a figura a seguir que retrata o conceito.

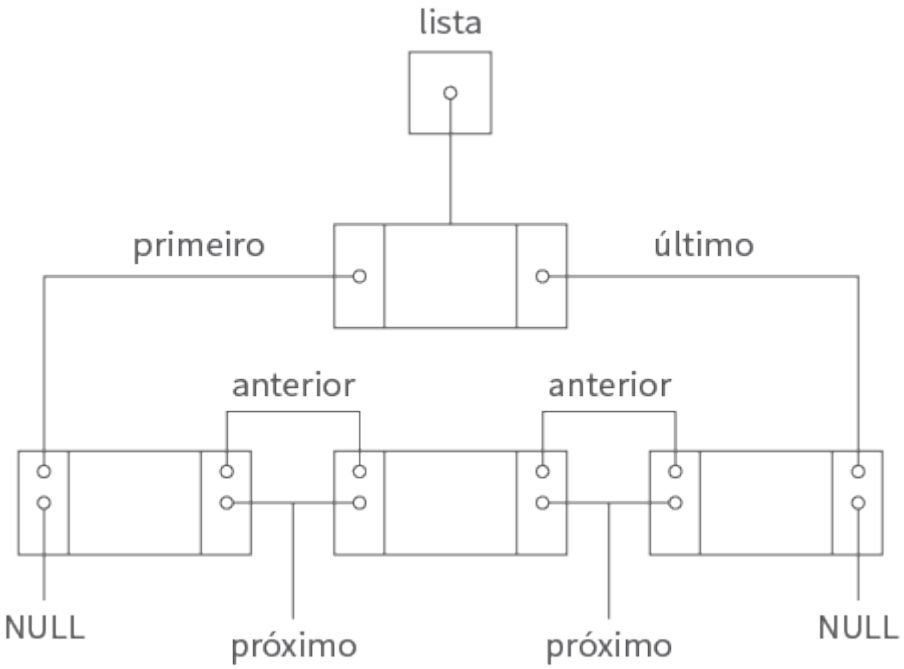

Figura 3 - Representação de uma lista duplamente encadeada
Fonte: Elaborada pela autora, 2019.

Para a representação em Java da lista duplamente encadeada, iremos acrescentar uma referência para o último item, na classe "lista", assim como também será adicionado um atributo do tipo "ItemLista" na classe de mesmo nome, representando a informação e uma referência (*ant*) para o elemento anterior do item. Observe a figura a seguir.

```
1  class Lista{
2      private ItemLista primeiroItem;
3      private ItemLista ultimoItem;
4  }
5
6  class ItemLista{
7       private String info;
8      private ItemLista prox;
9      private ItemLista ant;
10 }
```

Figura 4 - Implementação da lista duplamente encadeada em Java
Fonte: Elaborada pela autora, 2019.

Agora que já vimos o conceito de listas e os principais tipos, vamos estudar sobre as principais operações com essa estrutura de dados.

3.1.2 Operações com listas

A definição das operações em uma estrutura de dados sempre está atrelada ao tipo de dados armazenado. Assim, as principais operações de manipulação de uma lista se referem à leitura, inclusão e remoção de um item da lista, bem como o tamanho total de itens e a verificação se todos os elementos necessários estão presentes na lista.
A **inclusão** de um item pode ser implementada de formas diferentes. Pode-se optar por uma implementação em que o item sempre é inserido no final da lista ou incluir um item na lista, informando a posição em que se deseja inserir o elemento. Dessa forma, a definição da classe "lista", teria os seguintes métodos:

```
1  class Lista{
2          private ItemLista primeiroItem;
3        //métodos
4       public void adiciona(ItemLista novoItem){ }
5       public void adiciona(ItemLista novoItem, int posicao){
        }
6       public void remove(ItemLista itemParaRetirar){ }
7       public void ler(int posicao){ }
8       public int tamanhoLista(){ }
9      public boolean contem(ItemLista itemBuscado) { }
10 }
```

Figura 5 - Implementação das operações em uma lista encadeada
Fonte: Elaborada pela autora, 2019.

Independentemente de a lista ser simplesmente ou duplamente encadeada, as operações seriam as mesmas, pois a diferença está na implementação das operações. No caso de acionar um elemento na lista, temos duas assinaturas possíveis: adicionar um elemento sempre no início ou no final da lista; adicionar um item em uma posição específica. A operação de remoção, por sua vez, visitaria cada um dos elementos da lista até encontrar aquele correspondente ao item passado como parâmetro.

VOCÊ QUER VER?

Assim como tentamos buscar abstrair os conceitos para entender e fixar as estruturas de dados, a Inteligência Artificial é uma área da Ciência da Computação que busca implementar uma série de algoritmos e técnicas que imitam o raciocínio lógico adotado pelo ser humano. Esse é o tema do filme "Ela", dirigido por Spike Jonze. Trata-se de uma obra sobre um sistema operacional comprado por um homem solitário para lhe fazer companhia. Como o sistema é implementado de forma a sempre agradar o usuário com suas respostas, o homem acaba desenvolvendo uma relação amorosa com o sistema operacional.

No exemplo do “ItemLista”, o critério de comparação poderia ser o mesmo valor do atributo *info*. Ao ler um elemento da lista, deve ser passada a posição que ele ocupa. Observe que tal comportamento é o mesmo implementado quando acessamos um item em um vetor (*array*). O método que retorna o tamanho da lista é muito interessante, podendo ser utilizado tanto nos algoritmos de busca quanto nos algoritmos de ordenação: devemos visitar todos os itens da lista até encontrar o final da estrutura.

Crie duas classes Java denominadas “ItemLista” e “Lista”. Copie os códigos apresentados nas Figuras 2 e 5 e adicione um construtor na classe “Lista” que inicialize o atributo *prox* com o valor *null*. Implemente os métodos para adicionar e remover um item da lista. Lembre-se de que, para adicionar um elemento, você precisa criar um objeto do tipo “ItemLista” e, depois, fazer com o que atributo *prox* do elemento anterior aponte para o novo “ItemLista” criado. Para remover um item da lista, você tem que selecionar o elemento *prox* do elemento anterior e iniciá-lo com o valor do elemento apontado pelo atributo *prox* do elemento que está sendo removido.

Por fim, o método retorna um valor verdadeiro (*true*) ou falso (*false*), dependendo se o elemento passado como parâmetro foi encontrado na lista. Novamente, no exemplo do “ItemLista”, o critério de comparação poderia ser o mesmo valor do atributo *info*.

3.1.3 Aplicações práticas das listas

A aplicação prática de listas é diversificada, pois está, geralmente, associada a um conjunto de elementos (ZIVIANI, 2012).
Para conhecer mais sobre o tema, clique nas setas.
Pensando no contexto de um sistema empregado em uma instituição de ensino, em que precisamos de um *software* para gerenciar o corpo docente, as listas seriam utilizadas para representarem os alunos em uma classe.
Ao elaborar as listas de presença, os itens (alunos) são dispostos em ordem crescente do atributo de nome. Neste caso, é interessante observar como a estrutura de dados pode ser empregada pelos algoritmos de ordenação. O algoritmo de ordenação pelo nome, no caso, seria implementado dentro da classe “lista”.
Outra aplicação muito comum para as listas é na definição dos passos em um processo. Como vimos nas operações de um elemento do tipo “ItemLista”, ao adicionar um elemento, podemos optar pela sua inserção no final da lista ou em uma posição específica. A posição, nesse contexto, seria a ordem em que determinada tarefa deve ser realizada, sendo que a tarefa em si representa cada um dos itens da lista.
Agora que já conhecemos sobre as listas, vamos aprender outro conceito importante para nossa disciplina: a fila.

3.2 Fila

No dia a dia, estamos acostumados a vermos e enfrentarmos filas em diversos lugares: bancos, mercados, hospitais, cinemas, entre outros. Contudo, precisamos pensar que elas são essenciais, uma vez que determinam a ordem de atendimento das pessoas. Dessa forma, temos que os elementos de uma fila são acessados de acordo com a posição em que ocupam.

A construção de uma fila parte do princípio de que o primeiro a chegar é o primeiro a ser atendido. Nas filas dos supermercados, por exemplo, as pessoas são atendidas conforme a ordem de chegada, não é? Quando o primeiro da fila é chamado, a fila "anda", ou seja, o segundo passa a ser o primeiro, o terceiro passa a ser o segundo, e assim por diante, até chegarmos a última pessoa. Normalmente, para entrar em uma fila, uma pessoa deve se colocar na última posição, ou seja, no fim. Assim, quem chega primeiro tem prioridade.
A figura a seguir representa uma fila.

Figura 6 - Representação de uma fila
Fonte: Elaborado pela autora, 2019.

Conforme podemos observar na figura anterior, a estrutura fila se assemelha muito com a estrutura lista. O grande detalhe está nas operações de inserção (enfileirar) e remoção de um item (desenfileirar), que devem ocorrer de acordo com a ordem indicada pelos números 1, 2, 3 ..., *n*. Dizemos, neste caso, que as filas têm operações mais restritas que as listas.

3.2.1 Representação e operações das filas

A forma mais simples de implementar uma fila é por meio da representação de vetores (*arrays*). Diferentemente da lista, não há a necessidade de se inserir um elemento em uma posição diferente do final da fila (ZIVIANI, 2012). Assim, devemos manter apenas a posição do próximo elemento da fila para inserir um novo.
Já para desenfileirar um elemento, acessamos sempre o primeiro elemento da fila. Em seguida, precisamos atualizar a fila, passando o segundo elemento para a primeira posição, o terceiro elemento para a segunda posição, e assim sucessivamente.
Além do fato de sempre ter que atualizar os elementos em caso de retirada de um item da fila, essa representação é limitada, também, pelo tamanho inicial definido para o vetor. Se for necessário inserir um elemento a mais que o tamanho inicialmente definido, torna-se necessário declarar um novo vetor com tamanho maior e copiar os valores já inseridos.
Pensando no exemplo de implementar uma fila de pessoas, podemos definir a fila, os itens da fila e as operações necessárias conforme o código retratado na figura a seguir.

```
1  class FilaPessoas{
2      private Pessoa primeiraDaFila;
3      private Pessoal ultimaDaFila;
4       //métodos
5         public void enfileira(Pessoa novaPessoa){ }
6         public void desenfileira(Pessoa umaPessoa){ }
7         public void obtemPosicao(Pessoa umaPessoa){ }
8         public boolean estaVazia(){ }
9  }
10 class Pessoa{
11          private String nome;
12          private Pessoa prox;
13 }
```

Figura 7 - Implementação de uma fila de pessoas em Java
Fonte: Elaborada pela autora, 2019.

No código, uma fila tem uma referência para o primeiro e para o último elemento. Isto é importante porque a operação de enfileirar precisa da informação de quem é o último item da fila; enquanto a operação de desenfileirar necessita da referência para o primeiro elemento. A operação *obtemPosicao* retorna à posição ocupada pela pessoa passada como parâmetro. Por fim, a operação *estaVazia* retorna verdadeiro (*true*) caso a fila esteja vazia, sendo que, do contrário, retorna falso (*false*).

Crie duas classes Java denominadas “Pessoa” e “FilaPessoa”. Copie o código apresentado na Figura 7. A classe “Pessoa” deve ter dois atributos do tipo *string* (nome e CPF) e um atributo do tipo “Pessoa” com o valor *prox*. Adicione um construtor na classe “FilaPessoas” que inicialize os atributos *primeiraDaFila* e *ultimoDaFila* com o valor *null*. Este caso reflete a fila no início do expediente da instituição em que a fila de pessoas é formada, ou seja, a fila inicialmente está vazia. Implemente, então, os métodos para enfileirar e desenfileirar um item da fila. Lembre-se de que, para adicionar um elemento, você precisa criar um objeto do tipo “Pessoa” e, depois, fazer com o que atributo *prox* do elemento anterior aponte para o novo objeto do tipo “Pessoa” criado. Para desenfileirar, você tem que selecionar o elemento *primeiraDaFila* e iniciá-lo com o valor do elemento apontado pelo atributo *prox* do elemento que está sendo desenfileirado.

Na sequência, vamos conhecer algumas aplicações práticas das filas.

3.2.2 Aplicações práticas das filas

As aplicações práticas de filas são inúmeras. Existem filas com tamanho fixo (estáticas) e filas com tamanho indeterminado (dinâmicas).

Além disso, temos um tipo especial de fila estática denominada “circular dinâmica”, comumente utilizada pelos sistemas operacionais. Elas são usadas pelos algoritmos da memória principal do computador, a fim de realizarem o gerenciamento do que deve ou não estar na memória de acesso mais rápido e no relógio da máquina (DEVMEDIA, 2012). Como elas são circulares, o tamanho é fixo e os elementos inseridos na fila são alternados. Quando um elemento sai da fila, abre uma posição para a inserção de um novo elemento. Clique nos itens e conheça mais sobre o assunto.

Um exemplo muito comum de uso na área de sistemas computacionais e web são as filas de processos assíncronos. Estas, geralmente, são filas dinâmicas, já que não é possível determinar o tamanho exato de processos que serão enfileirados. Um processo é chamado de assíncrono quando não há uma resposta imediata ao ser realizada uma chamada de um serviço.
É muito comum quando o processamento é complexo ou quando depende da consulta a outros sistemas, como o processo de compra de um item em uma loja virtual. Após selecionarmos os itens de interesse e iniciarmos o processo de compra, o sistema precisa confirmar os dados de pagamento, verificar se o item está em estoque, separar o item para envio, enviar e, por fim, acompanhar a entrega do item até o destinatário. Por serem várias etapas que não são finalizadas de imediato, o uso de filas auxilia o procedimento.
Outro exemplo comum de fila dinâmica é quando vamos abrir um processo em um órgão público para tratar de alguma questão burocrática, como a emissão de uma certidão negativa de débitos ou o pedido de reavaliação de impostos cobrados.
Por ser um processo que demanda o acesso à base de dados, análise caso a caso e, talvez, o acesso a outros sistemas; geralmente, abrimos o processo e recebemos um identificador para consulta posterior.

VOCÊ SABIA?

A Java Message Service, especificação JSR-914, tem como objetivo o processo de troca de mensagens entre sistemas. Ela define dois padrões para o armazenamento de mensagens: *queue* (fila) e *topic* (tópico). A troca de mensagens pode ser usada em projetos de integração entre aplicações, sistemas de sincronização de banco de dados ou para um sistema de *chat* (DEVMEDIA, 2012).

Apesar de termos APIs e *frameworks* que já implementam grande parte das operações de filas, é importante entender o funcionamento para que as falhas eventuais que vierem a acontecer sejam identificadas com maior rapidez e acurácia.
Outro conceito essencial para nossos estudos é o de pilha. Vamos conhecer sobre ele e entender suas aplicações no tópico a seguir. Acompanhe!

3.3 Pilha

O conceito de pilha está igualmente relacionado ao termo utilizado em nosso dia a dia. Assim, temos a pilha de livros, a pilha de pratos, a pilha de processos, entre tantas outras.
Outro cenário comum para o uso do conceito é em uma fábrica de produção ou montagem de produtos. Determinado produto é composto por inúmeras peças (p1, p2, ..., *pn*), sendo que seu processo de montagem é automático (executado por uma máquina) e exige que as peças sejam colocadas em ordem específica (primeiro a

p1, depois a p2, depois a p3, e assim por diante). As peças, então, são empilhadas na ordem adequada e a máquina de montagem vai retirando peça por peça do topo da pilha para montar o produto final.
A figura a seguir representa uma pilha.

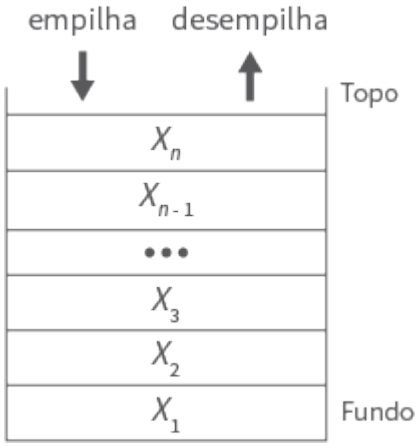

Figura 8 - Representação de uma pilha
Fonte: Elaborada pela autora, 2019.

Conforme pode ser identificado na figura anterior, a pilha tem uma definição muito similar ao de uma fila, com a diferença de que a inserção e a remoção de elementos ocorrem a partir de um mesmo ponto: o topo da pilha.
No item a seguir, veremos a representação e as principais operações dessa estrutura.

3.3.1 Representação e operações das pilhas

De forma similar a uma fila, a pilha poderia ser implementada como um vetor (*array*), mas teríamos as mesmas limitações: no caso de aumento da pilha, teríamos que declarar um novo vetor e copiar os valores para o novo (ZIVIANI, 2012). Além disso, ao retirar ou ao acrescentar um item na pilha, todos os elementos já existentes teriam que ser movidos. Para alguns objetos e ambientes, essas operações poderiam ser proibitivas.
Assim, utilizando o conceito de classe, a figura a seguir traz uma representação sugerida para a pilha e suas operações. A pilha implementada retrata a ordem de entrega de *pizzas*, de acordo com a rota definida previamente.

```
1  class PilhaEntregaPizza{
2      private Pizza topoPilha;
3       //métodos
4         public void  insereTopo(Pizza novaPizza){ }
5         public Pizza retiraTopo(){ }
6         public boolean estaVazia(){ }
7  }
8  class Pizza{
9          private String nomeDestinatario;
10         private String enderecoDestinatario;
11        private Pizza prox;
12 }
```

Figura 9 - Implementação de uma pilha de processos em Java
Fonte: Elaborada pela autora, 2019.

Temos, aqui, que a pilha é representada pela classe "PilhaEntregaPizza". Há a necessidade de se manter a referência para o topo da pilha, já que tanto a inserção quanto a retirada ocorrem pelo topo. As *pizzas*, então, terão que ser armazenadas no compartimento de entrega na ordem inversa. Assim, quando o entregador for realizar a entrega, as *pizzas* já estarão organizadas de forma que a mais próxima ao topo será entregue antes que aquelas mais distantes do topo. A terceira operação definida é a que verifica se a pilha está vazia. Cada pizza é representada por dois atributos: nome do destinatário e endereço de entrega.

Crie duas classes Java denominadas "Pizza" e "PilhaEntregaPizza". Copie o código apresentado na Figura 9. A classe "Pizza" deve ter dois atributos do tipo *string* (nome e endereço do destinatário da pizza) e um atributo do tipo *prox* que aponta para o próximo elementos da "Pilha". Adicione um construtor na classe "PilhaEntregaPizza" que inicializa o topo da pilha com o valor *null*. Implemente os métodos para "inserirTopo" (empilhar) e "removerTopo" (desempilhar) um item da pilha. Lembre-se de que, para adicionar ou remover um elemento na pilha, você precisa acessar o elemento apontado pelo atributo *topoDaPilha*. Para empilhar um novo elemento, devemos criar um objeto do tipo "Pizza", fazer com o atributo *prox* da nova "Pizza" aponte para o elemento no *topoDaPilha* atualmente e, depois, fazer com o que atributo *topoDaPilha* aponte para o novo objeto do tipo "Pizza" criado. Para desempilhar uma "Pizza" da pilha você tem que selecionar o elemento apontado pelo *topoDaPilha* e fazer com que este aponte para o valor apontado pelo atributo *prox* desse primeiro elemento.

Na sequência, vamos conhecer algumas aplicações práticas das pilhas.

3.3.2 Aplicações práticas das pilhas

O uso de pilhas é bastante comum em algoritmos que realizam a avaliação de expressões matemáticas. Por exemplo, na notação denominada "polonesa", os operadores aparecem imediatamente antes dos operandos. Tal notação define quais operadores e em que ordem devem ser calculados. Assim, dispensa o uso de parênteses, sem ambiguidades.

Digamos, por exemplo, que precisamos avaliar a expressão A * B - C / D. Usando a notação polonesa, a mesma expressão seria - * A B / C D. A notação dita "polonesa reversa" (*posfix*) é como a polonesa, em que os operandos aparecem após os operadores. No exemplo, a expressão seria A B * C D / -. Poderíamos utilizar uma estrutura do tipo pilha e inserir os elementos na ordem polonesa reversa. Dessa forma, inicialmente, seriam desempilhados os valores de A e B. Ao desempilhar o operador *, seria feita a multiplicação A*B. Em seguida, seriam desempilhados os valores de C e D. Em seguida, ao ler o operador /, seria feita a divisão de C por D. Ao final, é desempilhado o operador -. Assim, o resultado A*B é decrementado do resultado de C/D.

VOCÊ SABIA?

A execução de um sistema computacional, independentemente da plataforma, funciona sobre o conceito de pilha de execução. Uma pilha de execução ou pilha de chamada armazena informações sobre as sub-rotinas ativas em um programa de computador. Seu principal uso é registrar o ponto em que cada sub-rotina ativa deve retornar o controle de execução quando terminar de executar. Quando estamos depurando um código, estamos montando essa pilha mentalmente, pois a ideia é visualizar o trajeto que o fluxo de execução do programa fez.

Na próxima seção veremos um exemplo prático da aplicação de pilhas na API da linguagem Java. Acompanhe com atenção!

3.3.3 Implementação de pilhas com API Java

A API do ambiente de execução Java contém uma série de bibliotecas que estão disponíveis para serem utilizadas pelos sistemas desenvolvidos com essa tecnologia. Entre as classes disponíveis, temos a *stack*, que implementa uma pilha. Ela implementa o mecanismo LIFO (*last in*, *first out*), em que o último elemento a entrar na pilha é o primeiro a sair. Em outras palavras, tanto a inserção quanto a retirada de elementos ocorrem pelo topo da pilha.

A figura abaixo foi extraída da documentação da classe *stack*.

Figura 10 - Hierarquia e definição do pacote em que a classe stack está inserida
Fonte: Elaborada pela autora, 2019.

Conforme apresentado na figura anterior, a classe *stack* está inserida dentro do pacote *java.util*. Assim, para utilizá-la, é necessário usar, explicitamente, a diretiva *import*, seguida do nome do pacote: *import java.util;*.
Além disso, a figura também nos informa que a classe *stack* estende a classe *vector* com cinco operações que viabilizam que um vetor trabalhe como uma pilha. Essas cinco operações são: inserir um elemento na pilha ou empilhar (*push*), remover o elemento do topo da pilha ou desempilhar (*pop*), ler o elemento que está no topo da pilha (*peek*), verificar se a pilha está vazia (*empty*) e buscar por um elemento na pilha e retornar o quão longe ele está do topo (*search*). O topo da pilha é definido como sendo a posição de número 1 e a operação retorna à posição do primeiro elemento encontrado.
Para criar uma pilha temos que criar um objeto dessa classe: Stack pilha = new Stack(). Quando uma pilha é criada, ela está, inicialmente, vazia. A função *push*() recebe como parâmetro um *object*, portanto, para armazenar primitivos, será necessário armazenar uma classe *wrapper* (*integer*, *double*, *float* etc.). As funções *pop*() e *search* () retornam um objeto.

CASO

Vamos criar como exemplo uma pilha de elementos do tipo inteiro. Os valores serão gerados aleatoriamente com o auxílio da classe *random*. A partir de um objeto dessa classe, é possível gerar um valor inteiro entre 0 e o número informado como parâmetro para o método *nextInt* (). Vejamos a trecho de código a seguir:

```
Stack pilha = new Stack(); // cria um objeto da classe Stack
 // cria objeto que permitirá a geração dos números
aleatórios
    Random random = new Random();
//laço for executa 10 vezes
    System.out.println(">> Inseridos na pilha os valores: );
    for (int i=0;i<10;i++){
// Insere na pilha números aleatórios entre 0 a 100
        System.out.println((i+1)+" :"+
        pilha.push(random.nextInt(100)));
     }
// Retira da pilha os 10 elementos
 System.out.println("<< Removidos da pilha os valores: );
    for (int i=0;i<10;i++){
        System.out.println((i+1)+" : " + pilha.pop());
    }
```

A saída para o código anterior está apresentada na figura na sequência.

```
>> Inseridos na pilha os valores:
1: 17
2: 21
3: 14
4: 18
5: 84
6: 56
7: 78
8: 65
9: 34
10: 46
<< Removidos da pilha os valores:
1: 46
2:34
3:65
4: 78
5: 56
6: 84
7: 18
8: 14
9: 21
10: 17
```

Observe que, conforme o esperado, a ordem de retirada é exatamente o inverso da ordem de inserção. Se você executar esse código na sua máquina, a saída será diferente, pois o gerador de números aleatórios irá se encarregar da exclusividade dos valores numéricos na pilha gerada.

Dessa forma, temos que as estruturas de dados são importantes elementos para a construção de sistemas computacionais, independentemente, da plataforma. A escolha correta dessa estrutura vai proporcionar ao programador as ferramentas certas para a manipulação dos dados. É importante, também, observar que os conceitos estão bem atrelados ao uso dos termos em nosso cotidiano. Isso nos auxilia quanto ao entendimento da estrutura.

Síntese

Chegamos ao fim de mais uma unidade. Aqui, estudamos a respeito dos três tipos básicos de estruturas de dados: listas, filas e pilhas. Vimos que a forma de armazenamento e as operações disponíveis para cada uma delas está diretamente relacionada com o comportamento esperado da aplicação. Além disso, também pudemos aprender quanto a um exemplo prático de pilhas na API.

Nesta unidade, você teve a oportunidade de:

- compreender que as estruturas de dados lista, fila e pilha se diferenciam, especialmente, pela ordem com que os elementos são inseridos e retirados em cada estrutura;
- identificar a diferença entre uma estrutura de dados estática e dinâmica, sendo que cada tipo tem vantagens e desvantagens de uso;
- entender que os conceitos, apesar de teóricos, tentam imitar as aplicações práticas do dia a dia;
- conhecer algumas bibliotecas da linguagem Java que já implementam aplicações comuns dessas estruturas para finalidades específicas.

Bibliografia

ASCENCIO, A. F. G.; ARAÚJO, G. S. **Estrutura de dados**: algoritmos, análise da complexidade e implementações em Java e C/C++. São Paulo: Pearson, 2010.

DEVMEDIA. **Como implementar a troca de mensagens com JMS**. 2012. Disponível em:https://www.devmedia.com.br/como-implementar-a-troca-de-mensagens-com-jms/25127. Acesso em: 30 jul. 2019.

DEVMEDIA. **Fila circular dinâmica**. 2012. Disponível em: https://www.devmedia.com.br/fila-circular-dinamica/24572. Acesso em: 30 jul. 2019.

PUGA, S.; RISSETTI, G. **Lógica de programação e estruturas de dados – Com aplicações em Java**. 3. ed. São Paulo: Pearson, 2010.

ROBERTS, S. Conheça o cientista considerado o 'guia espiritual' dos algoritmos. **Estadão**, Nova York, 2018. Disponível em: https://internacional.estadao.com.br/noticias/nytiw,conheca-o-cientista-considerado-o-guia-espiritual-dos-algoritmos,70002654893. Acesso em: 30 jul. 2019.

TAMASSIA, R.; GOODRICH, M. T. **Estruturas de dados e algoritmos em Java**. Porto Alegre: Grupo A, 2011.

ZIVIANI, N. **Projeto de algoritmos**: com implementações em JAVA e C++. 3. ed. São Paulo: Cengage Learning Editores, 2012.