ESTRUTURA DE DADOS

UNIDADE 4 – ÁRVORES DE DECISÃO

Katia Cristina Lage dos Santos

Introdução

Certamente você já teve que preencher algum questionário ou formulário que, ao final das respostas, é fornecido um resultado de acordo com as opções marcadas, não é? Outra situação similar é quando você chega para um atendimento e a enfermeira que faz a triagem realiza uma série de perguntas para identificar qual é a sua prioridade no atendimento. Esse tipo de mecanismo que realiza uma decisão com base nas respostas é chamado de mecanismos de decisão. Um tipo abstrato de dados comumente utilizado para representar esse mecanismo é denominado árvore de decisão. Será que essa estrutura de dados tem alguma analogia com o conceito de árvore na vida real?

Assim como os demais tipos abstratos de dados, as árvores de decisão têm, sim, uma analogia ao conceito de árvore real. Elas são formadas por folhas e galhos que ligam os nós. Assim, se esse é outro tipo abstrato de dado, então temos uma nova forma de armazenar os dados e as operações relacionados ao conceito? Existe apenas um tipo ou diferentes tipos de árvores de decisão?

Ao longo desta unidade, veremos sobre a forma de armazenamento, as operações e dois tipos de árvores de decisão: as árvores binárias e AVL. Entenderemos que os dois modelos têm uma característica em comum: a partir de um mesmo nó pai, ambas têm, no máximo, dois nós filhos que estão diretamente ligados a ele. Serão apresentados exemplos de como armazenar uma informação em um nó de uma árvore e como relacionar um nó com seus filhos. Em termos de operações sobre esses dados, serão vistas as operações de leitura, inserção, remoção e busca por um nó em uma árvore, com a implementação de cada um dos métodos. No caso da busca, veremos três métodos de caminhamento distintos.

Vamos, então, iniciar nossos estudos!

4.1 Árvores binárias

Em termos de representação, a melhor forma de comparar uma árvore de decisão ao conceito que conhecemos na vida real é pensar em uma árvore invertida. Geralmente, a raiz da planta fica em sua base, na parte mais baixa, sendo que, a partir dela, crescem os galhos, que, por sua vez, têm várias folhas.

A figura a seguir retrata um tipo abstrato de dados no modelo árvore.

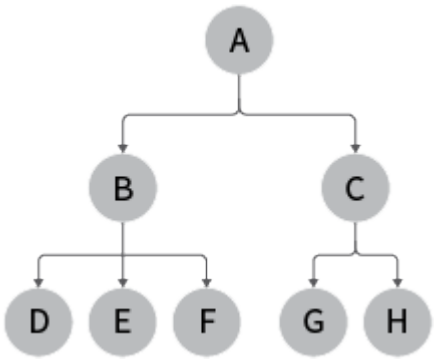

Figura 1 - Representação gráfica de uma árvore de decisão
Fonte: Elaborada pela autora, 2019.

Na árvore da figura anterior, cada círculo com uma letra é denominado “nó” ou “célula”. O nó A é chamado de “nó raiz”, pois dele derivam os demais nós. Dizemos que o nó raiz está no nível 0 da árvore.

Cada um dos nós ligados diretamente ao nó A é chamado de “nó filho de A”, ou seja, B e C. Eles, por sua vez, estão no nível 1 da árvore, ligados por meio de arestas ou galhos.
Os nós D, E, F, G e H são chamados de nós filhos de B e C, sendo estes chamados de “nós folhas”, pois deles não derivam outros nós na árvore. Observe que não tem uma aresta que sai desses nós, apenas aresta que chega de B ou C. No caso da nossa representação, os nós folhas estão no nível 2. Por fim, dizemos que a árvore tem altura 2, pois existem dois níveis que separam o nó raiz dos nós folhas.

VOCÊ O CONHECE?

Alan Mathison Turing foi um matemático, lógico, criptoanalista e cientista da computação, influente no desenvolvimento da Ciência da Computação e na formalização do conceito de algoritmo e computação com a máquina de Turing. Em especial, também foi o responsável pela criação do Teste de Turing, mecanismo que verifica a capacidade de um *software* se comportar de forma similar aos seres humanos (OLIVEIRA, 2019).

Uma árvore binária tem como característica fundamental todos os nós que não sejam nós folhas com um ou dois nós filhos (PUGA; RISSETTI, 2010). Daí vem o nome da árvore.
Vejamos a figura a seguir, que representa uma árvore binária bem similar à árvore anterior. A diferença é que, nesta, os nós B e C têm exatamente dois filhos cada um.

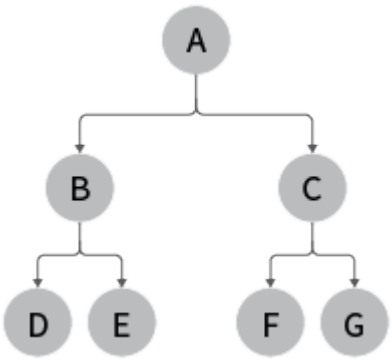

Figura 2 - Representação gráfica de uma árvore binária
Fonte: Elaborada pela autora, 2019.

No item a seguir, veremos a forma de armazenamento e as operações que podem ser realizadas na árvore de decisão. Acompanhe!

4.1.1 Armazenamento de dados em uma árvore binária

Um nó, em uma árvore, armazena informações e referências para os filhos da esquerda e da direita. Esses elementos devem ser declarados como atributos de uma classe. A figura a seguir apresenta a definição de um nó genérico.

```
class NoArvoreBinario{
    private String informacao;
    private NoArvoreBinario filhoEsquerda;
    private NoArvoreBinario filhoDireita;
}
```

Figura 3 - Representação de um nó na árvore binária
Fonte: Elaborada pela autora, 2019.

Na definição da classe “NoArvoreBinario”, a informação será armazenada no primeiro atributo do tipo *string*. Se este for um nó folha, os valores de “filhoEsquerda” e “filhoDireita” será *null*. Caso contrário, cada um dos nós referenciará outro “NoArvore” correspondente ao filho da esquerda ou da direita. Clique nas setas e aprenda mais sobre o tema.

No entanto, o que acontece se a informação que precisa ser armazenada na árvore possui um tipo mais complexo que não pode ser guardado como um *string*? Neste caso, quando a informação que precisa ser armazenada no tipo abstrato de dados (TAD) tem várias características, criamos uma classe e, nela, encapsulamos os atributos.

Vamos considerar o seguinte cenário: criaremos uma representação da árvore genealógica em que cada pessoa tem, no máximo, dois filhos. Isto é, considerando os pais, avós, filhos e netos, todos os nós respeitam a propriedade para que possamos construir uma árvore binária. Assim, cada indivíduo tem um conjunto de informações que o define, por isso, usaremos um tipo complexo para definir ou encapsular esse dado.

Inicialmente, observe na figura a seguir o trecho de código que contém a definição de uma classe “Pessoa”. Segundo tal definição, cada pessoa tem três informações: nome, CPF e data de nascimento.

```
class Pessoa{
    private String nome;
    private String cpf;
    private Date dataNascimento;
    public String getNome(){
        return nome;
    }
    public void setNome(String nome){
        this.nome = nome;
    }
    public String getCpf(){
        return cpf;
    }
    public void setCpf(String cpf){
        this.cpf = cpf;
    }
    public Date getDataNascimento(){
        return dataNascimento;
    }
    public void setDataNascimento(Date data){
        dataNascimento = data;
    }
}
```

Figura 4 - Classe "Pessoa"
Fonte: Elaborada pela autora, 2019.

Veja que, como os atributos são privados, foram criados métodos públicos para fazer a leitura de cada um deles, chamados de métodos *get*. Para alterar o valor dos atributos, foram criados métodos *set*.
Agora, vejamos a figura na sequência.

```
class NoArvoreBinario{
    private Pessoa pessoa;
    private NoArvoreBinario filhoEsquerda;
    private NoArvoreBinario filhoDireita;
}
```

Figura 5 - Nó de uma árvore cujos nós são “Pessoas”
Fonte: Elaborada pela autora, 2019.

A classe “NoArvore”, relacionada ao novo contexto, deve ter a informação como sendo um objeto do tipo “Pessoa”, conforme vimos na figura anterior.

Agora, faremos a representação de uma árvore binária cujos elementos são do tipo “Fruta”. Para isso, crie duas classes Java: “Fruta” e “NoFruta”.

A classe “Fruta” terá três atributos: nome, cor e tipo. Declare como atributos privados e crie os métodos *get* e *set* para cada atributo.

A classe “NoFruta” será similar à classe “NoArvore”, declarada anteriormente, sendo que a diferença estará no tipo de informação, que, no caso, será do tipo “Fruta”. O nome da variável de referência deste será “fruta” e os filhos da esquerda e da direta deverão ter, respectivamente, os nomes “frutaEsquerda” e “frutaDireita”.

Agora que já vimos a forma de armazenamento das informações em uma árvore, veremos as operações sobre esse tipo de dado.

4.1.2 Operações de leitura e inserção de nós em árvores binárias

Como a árvore é um tipo abstrato de dados e já definimos a forma de armazenamento destes, precisamos, agora, definir as operações sobre os dados armazenados.
Em linhas gerais, temos que criar as operações descritas abaixo. Clique e confira!
Ler a informação de nó de uma árvore.
Inserir ou alterar um nó em uma árvore.
Verificar se a árvore está vazia.
Remover um nó de uma árvore.
Buscar um elemento na árvore.
Inicialmente, vamos tratar da leitura e inserção/alteração de um elemento na árvore binária. Tanto a leitura quanto a inserção pode ser referente ao filho da esquerda ou da direita. Para
Depois que encapsulamos as informações da pessoa em uma classe, precisamos criar a representação de um nó dessa árvore genealógica. Na figura a seguir, temos o trecho de código em que a informação é do tipo “Pessoa”.

```
class NoArvore{
    private Pessoa pessoa;
    private NoArvore filhoEsquerda;
    private NoArvore filhoDireita;
    public Pessoa getPessoa(){
        return pessoa;
    }
    public void setPessoa(Pessoa pessoa) {
        this.pessoa = pessoa;
    }
    public NoArvore getFilhoEsquerda(){
        return filhoEsquerda;
    }
    public void setFilhoEsquerda(NoArvore filhoEsquerda)
    {
        this.filhoEsquerda = filhoEsquerda;
    }
    public NoArvore getFilhoDireita(){
        return filhoDireita;
    }
     public void setFilhoDireita(NoArvore filhoDireita)
    {
        this.filhoDireita = filhoDireita;
    }
}
```

Figura 6 - Nó de uma árvore com informações sobre uma pessoa
Fonte: Elaborada pela autora, 2019.

Observe que, nesse caso, a informação é o objeto do tipo "Pessoa". Como este atributo e as referências para os filhos da esquerda e da direita são privados, foram criados métodos públicos *get* e *set* para, respectivamente, fazer a leitura e inserção/alteração do valor desses atributos. Para começar, as referências para os filhos da esquerda e direita são nulas, pois não estão referenciando um objeto do tipo "NoArvore".
Assim, na sequência, precisamos criar a árvore em si. Vamos considerar uma parte da árvore ou subárvore em que José é o pai e Francisco e Maria são os filhos.

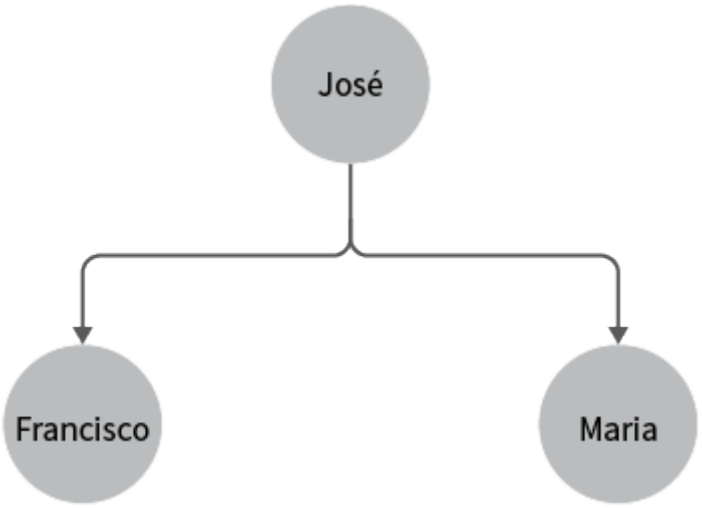

Figura 7 - Representação de uma subárvore de José e seus filhos
Fonte: Elaborada pela autora, 2019.

O trecho de código correspondente à criação do primeiro nó da árvore, também conhecido como nó raiz. Este é do tipo "NoArvore", já que também contém informações da árvore genealógica. Para preencher a informação sobre a primeira pessoa mapeada na família, foi criado um objeto do tipo pessoa, sendo "primeiraPessoa" o nome da variável que referencia o primeiro objeto. Além disso, o método *set* foi invocado para preencher os atributos da primeira pessoa: nome, CPF e data de nascimento. Posteriormente, foram associados os dados da primeira pessoa como sendo o elemento raiz da árvore, por meio do método "setPessoa" da classe "NoArvore". Veja a figura na sequência.

```
class Arvore{

    public static void main()
    {
        NoArvore raiz = new NoArvore();
        Pessoa primeiraPessoa = new Pessoa();
        primeiraPessoa.setNome("José Araújo");
        primeiraPessoa.setCpf("001112223-44");
        primeiraPessoa.setDataNascimento(new Date(1811, 10, 10));

        raiz.setPessoa(primeiraPessoa);
    }

}
```

Figura 8 - Declaração de uma árvore com preenchimento do nó raiz
Fonte: Elaborada pela autora, 2019.

Precisamos, então, criar os nós correspondentes aos filhos de José e atribuir os valores aos nós da esquerda e da direita. O trecho de código da figura na sequência nos mostra essa implementação de modo ilustrativo.

```
class ArvoreGenealogica{

    private NoArvore raiz;

    public static void main()
    {
        raiz = new NoArvore();
        Pessoa primeiraPessoa = new Pessoa();
        primeiraPessoa.setNome("José Araújo");
        primeiraPessoa.setCpf("001112223-44");
        primeiraPessoa.setDataNascimento(new Date(1811, 10, 10));

        raiz.setPessoa(primeiraPessoa);

        Pessoa francisco = new Pessoa();
        francisco.setNome("Francisco Araújo");
        francisco.setCpf("998887665-31");
        francisco.setDataNascimento(new Date(1830, 06, 01));
        raiz.setFilhoEsquerda(francisco);

        Pessoa maria = new Pessoa();
        francisco.setNome("Maria Araújo");
        francisco.setCpf("223334445-76");
        francisco.setDataNascimento(new Date(1836, 02, 25));
        raiz.setFilhoDireita(maria);
    }
```

Figura 9 - Implementação da subárvore de José e seus filhos
Fonte: Elaborada pela autora, 2019.

Para criar os nós da árvore correspondente aos filhos “Francisco” e “Maria”, inicialmente, foi criado um objeto do tipo “NoArvore” para cada um deles. A inserção do nó filho da esquerda foi enviada à variável de referência para o objeto criado anteriormente como parâmetro do método “setFilhoEsquerda()” a partir do nó raiz. De forma similar, para inserir o nó filho da direita, foi necessário passar a referência do nó correspondente à filha “Maria” para o método “setFilhoDireita()” a partir do nó raiz.

Para verificar se uma árvore está vazia, é necessário analisar se a raiz da árvore não tem nenhuma referência para o elemento correspondente à informação da árvore. O método retorna *true* (verdadeiro) se a árvore está vazia e *false* (falso) se for o contrário. No exemplo da árvore genealógica, temos que verificar se o atributo raiz está apontando para o valor *null*.

O trecho de código da figura a seguir apresenta a implementação do método “estaVazia()”. Assim, se a informação for um tipo de dado primitivo, temos que verificar em qual valor padrão que a variável inicia ou iniciar o nó raiz com um valor pré-definido que indique que nenhum nó efetivo foi adicionado à árvore.

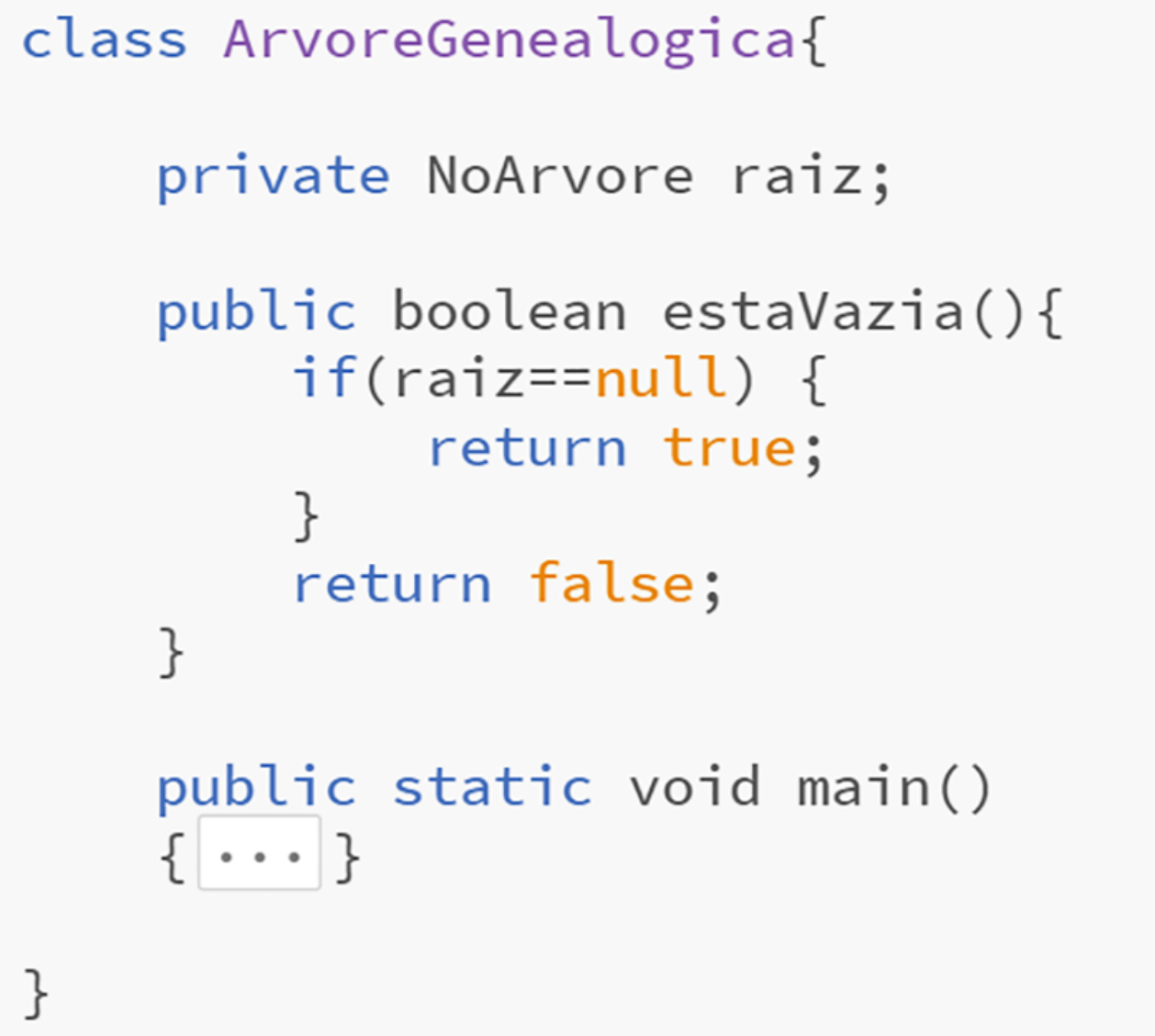

Figura 10 - Método que verifica se a árvore está vazia
Fonte: Elaborada pela autora, 2019.

Para realizar a busca em uma árvore, podemos implementar um mecanismo similar à busca sequencial, em que todos os nós da árvore são visitados até que a informação seja encontrada. Como alternativa para melhorar a *performance* da busca, podemos definir um critério para a inserção dos filhos da esquerda e da direita em um nó, com base em uma informação-chave, a fim de se criar o que chamamos de "busca binária de busca".

Vamos continuar a implementar a representação de uma árvore binária cujos elementos são do tipo "Fruta"? Para isso, utilize as duas classes criadas anteriormente: "Fruta" e "NoFruta".

Crie os métodos *get* e *set* de cada um dos atributos: "fruta", "frutaEsquerda" e "frutaDireita". Crie, também, um método para verificar se a árvore está vazia, retornando *true* em caso verdadeiro ou *false* do contrário.

Agora, crie uma classe "Arvore" e declare o método *main*. Declare um nó da classe "NoFruta" e faça uma chamada no método que verifica se a árvore está vazia. Em seguida, crie um objeto do tipo "Fruta" e passe a referência dele para o método "setFruta" da classe "NoFruta". Confira se os valores de "filhoEsquerda" e "filhoDireita" são iguais a *null*.

Por fim, crie frutas filhas da esquerda e da direita e teste os métodos *get* e *set* para confirmar o funcionamento do código.

Estudaremos mais sobre a árvore binária de busca e as diferentes formas de caminhamento ou percurso na próxima seção. Vejamos!

4.2 Árvore binária de busca

Uma árvore binária de busca é um tipo especial de árvore binária em que seguimos uma regra no momento de definir o nó filho da esquerda ou da direita. Para tanto, inicialmente, precisamos definir um atributo-chave do nó que será utilizado como comparação com os nós já inseridos. Assim, a regra das árvores binárias de busca diz que, para cada nó N da árvore, todos os nós na subárvore da esquerda de N têm chave menor que "N.chave"; e todos os nós na subárvore direita de N têm chave maior que "N.chave". Como a árvore é uma estrutura hierárquica, para atendimento dessa condição, temos que aplicá-la de forma recursiva até que o novo nó seja inserido e a constituição da árvore ainda seja mantida (ZIVIANI, 2012).

VOCÊ QUER VER?

O filme "O Jogo da Imitação", dirigido por Morten Tyldum, é baseado na história real do cientista Alan Turing. O longo mostra como a Ciência da Computação foi importante para a vitória dos aliados na Segunda Guerra Mundial, graças ao trabalho de um grupo de cientistas liderados por Alan Turing, após serem contratados pelo governo inglês para desvendarem o segredo da máquina "Enigma", utilizada pelos nazistas para comunicações sigilosas. Vale a pena se aprofundar na temática da obra!

Vamos considerar uma árvore cujo elemento-chave de cada nó seja uma letra do alfabeto. Além disso, devemos inserir os seguintes elementos na ordem definida: {S, C, A, X, R, Z}. Na figura a seguir, é apresentada a inserção de cada um dos nós passo a passo.

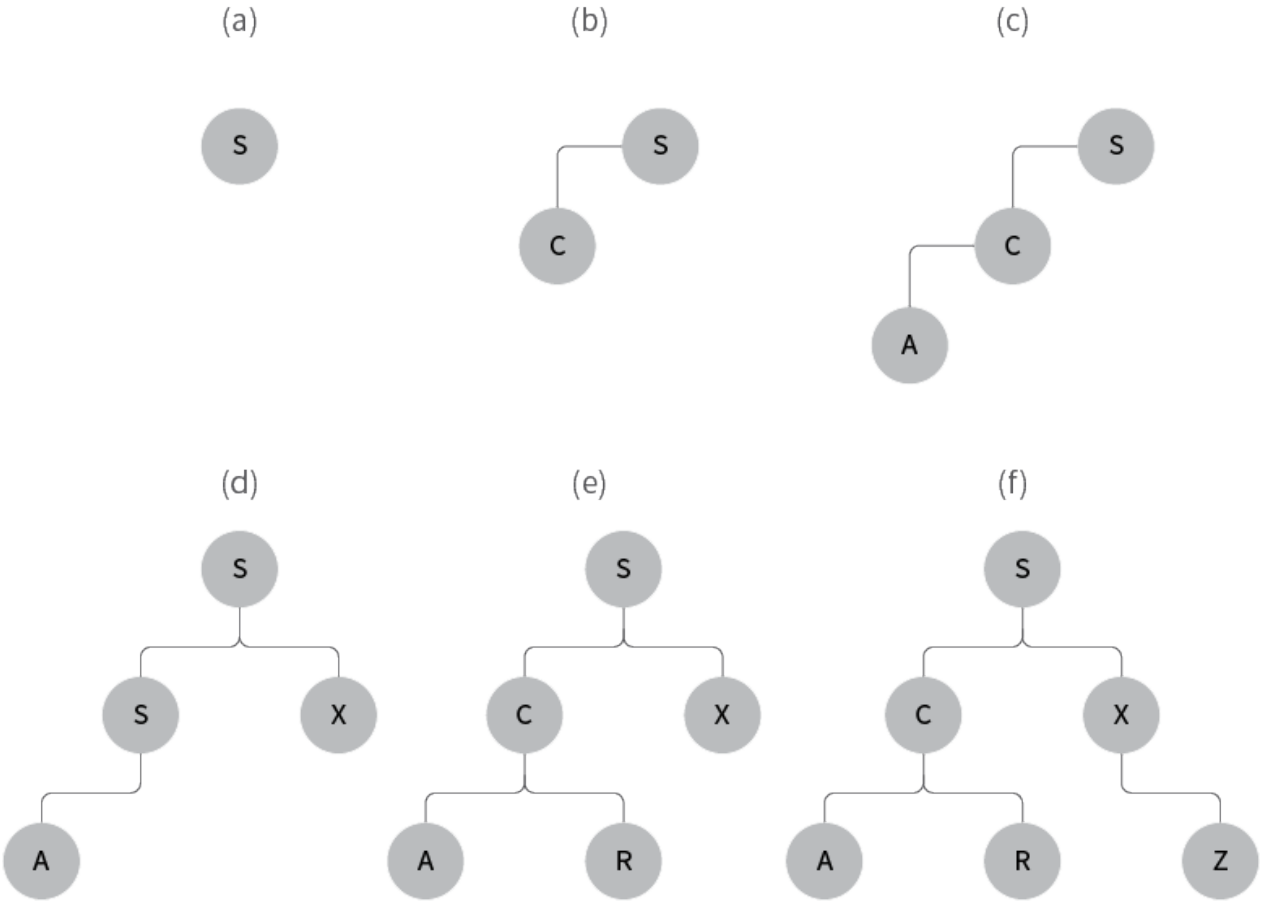

Figura 11 - Inserção de nó em uma árvore de busca binária
Fonte: Elaborada pela autora, 2019.

Agora, aprenda mais sobre a inserção de nó em uma árvore binária clicando nos itens abaixo.

Observe que o primeiro nó inserido (S) é o nó raiz da árvore, apresentado em (a).
Em seguida, o nó C deve ser inserido na árvore. Como, na ordem alfabética, a letra C vem antes da letra S e não tem um nó atrelado ao nó S à esquerda, o nó C é inserido como o filho à esquerda do nó S, apresentado em (b).
No passo (c), é inserido o nó A. Inicialmente, ele é comparado com S. Como este é menor que A, ele deve ser inserido no lado esquerdo. Como S já tem um filho à esquerda, A é comparado com C. Além disso, como A é menor que C e este é um nó folha (não tem nós filhos), A é inserido como filho à esquerda de C.
O próximo nó a ser inserido é X. Como ele vem após S, então é direcionado para a subárvore da direita. Como S não tem filho à direita, em (d), X é inserido como filho à direita de S.
O próximo nó a ser inserido é R. Então, em (e), R é comparado com S. Como R vem antes de S, ele é direcionado para a subárvore da esquerda.
Em seguida, é comparado com C. Como R é maior que C, é direcionado para a subárvore da direita. O nó R é, então, inserido à direita de C, pois não tem filho ocupando essa posição.
Por fim, deve ser inserido o nó Z. Ele é comparado com S. Pelo fato de Z ser maior que S, ele é direcionado para a subárvore da direita
Em seguida, Z é comparado com X. Como Z é maior que X, ele é inserido como filho à direita de X.

A implementação do algoritmo que realiza esse passo a passo de inserção de um novo nó em uma árvore de busca está apresentada na figura na sequência.

```
1  private No adicionaNo(No atual, char valor) {
2      if (atual == null) {
3          return new No(valor);
4      }
5
6      // valor a ser inserido é menor que o nó corrente
7      if (valor < atual.valor) {
8          atual.filhoEsquerda = adicionaNo(atual.filhoEsquerda, valor);
9      } else if (valor > atual.valor) {// valor a ser inserido é maior que corrente
10         atual.filhoDireita = adicionaNo(atual.filhoDireita, valor);
11     } else {
12         // valor já existe, não precisa adicionar novamente
13         return atual;
14     }
15     return atual;
16 }
```

Figura 12 - Método para inserção de um nó em uma árvore binária de busca
Fonte: Elaborada pela autora, 2019.

O método "adicionaNo" recebe dois parâmetros: nó corrente (atual) e valor a ser adicionado (valor). Observe que o método é recursivo, pois, dentro da sua definição, há uma chamada ao próprio método. O algoritmo, então, percorre a árvore até encontrar o lugar onde o novo nó deve ser inserido. Quando o lugar é encontrado, temos a situação em que o valor de atual é igual a *null*. Essa é, também, a condição de parada do algoritmo.
Enquanto não encontramos a posição em que o nó deve ser inserido, continuamos a busca, agora pelas subárvores. Como existem duas subárvores (esquerda e direita), para saber qual subárvore deve ser percorrida, é necessário fazer uma comparação do valor a ser inserido com aquele que está sendo visitado (atual). Nessa comparação, podemos redirecionar a busca para a esquerda ou direita por meio de uma chamada recursiva ao próprio método.

Utilizando o mesmo mecanismo apresentado na Figura 11, faça o esquema de inserção dos nós na árvore na ordem inversa: {Z, R, X, A, C, S}. Crie cada um dos seis passos, utilizando como referência o algoritmo apresentado anteriormente. Lembre-se de que é importante compreender o mecanismo de recursividade presente nesse algoritmo

Nos itens na sequência, veremos três métodos para realizar o percurso em uma árvore binária de busca. Tais métodos também são recursivos e comumente utilizados para a busca de um elemento em uma árvore binária de busca. Importante observar que eles assumem que a árvore foi construída de acordo com o algoritmo da última figura apresentada, pois, assim, foi adotada uma regra na inserção dos nós na árvore.

4.2.1 Pré-ordem

Percorrer ou realizar o percurso em uma árvore em pré-ordem significa fazer a busca sob a seguinte ordem:

- visitar o nó raiz da subárvore corrente;
- percorrer a subárvore da esquerda em pré-ordem;

- percorrer a subárvore da direita em pré-ordem.

Temos, ainda, que visitar um nó significa realizar algum tipo de ação sobre ele, como comparar o seu valor com um valor procurado.

A implementação do método que realiza a busca em pré-ordem está apresentada na figura a seguir. Recebemos como parâmetro o nó atualmente visitado e o nó com uma chave utilizada como elemento de comparação.

```
private String procuraPreOrdem(No atual, No procurado){
    if (atual.getChave().equalsIgnoreCase(procurado.getChave())) {
        return procurado.getValor();
    }

    if(atual.getFilhoEsquerda!=null)
    {
        procuraPreOrdem(no.getFilhoEsquerda());
    }

    if(atual.getFilhoDireita!=null)
    {
        procuraPreOrdem(no.getFilhoDireita());
    }

    return null;
```

Figura 13 - Busca em pré-ordem
Fonte: Elaborada pela autora, 2019.

Observe que, inicialmente, o nó raiz é passado como o nó atual do método “procuraPreOrdem”. Quando o elemento-chave é encontrado, podemos retornar o valor associado ao nó.

4.2.2 Ordem simétrica ou in-ordem

Percorrer ou realizar o percurso em uma árvore em ordem simétrica significa fazer a busca sob a seguinte ordem:

- percorrer a subárvore da esquerda em ordem simétrica;
- visitar o nó raiz da subárvore corrente;
- percorrer a subárvore da direita em ordem simétrica.

A implementação do método que realiza a busca em ordem simétrica está apresentada na figura na sequência.

```
private String procuraOrdemSimetrica(No atual, No procurado){
    if(atual.getFilhoEsquerda!=null)
    {
        procuraOrdemSimetrica(no.getFilhoEsquerda());
    }

    if (atual.getChave().equalsIgnoreCase(procurado.getChave())) {
        return procurado.getValor();
    }

    if(atual.getFilhoDireita!=null)
    {
        procuraOrdemSimetrica(no.getFilhoDireita());
    }

    return null;
```

Figura 14 - Busca em ordem simétrica
Fonte: Elaborada pela autora, 2019.

Na seção seguinte, veremos um método de busca em árvores binárias com uma priorização pela busca nas subárvores, em comparação com a pelo nó raiz.

4.2.3 Pós-ordem

Por fim, percorrer ou realizar o percurso em uma árvore em pós-ordem significa fazer a busca sob a seguinte ordem:

- percorrer a subárvore da esquerda em pós-ordem;
- percorrer a subárvore da direita em pós-ordem;
- visitar o nó raiz da subárvore corrente.

```
private String procuraPosOrdem(No atual, No procurado){
    if(atual.getFilhoEsquerda!=null)
    {
        procuraPosOrdem(no.getFilhoEsquerda());
    }

    if(atual.getFilhoDireita!=null)
    {
        procuraPosOrdem(no.getFilhoDireita());
    }

    if (atual.getChave().equalsIgnoreCase(procurado.getChave())) {
        return procurado.getValor();
    }

    return null;
```

Figura 15 - Busca em pós-ordem
Fonte: Elaborada pela autora, 2019.

A implementação do método que realiza a busca em pós-ordem está apresentada na figura anterior, como pudemos observar.

CASO

Como pudemos ver, a ação de visitar um nó durante o percurso em uma árvore binária de busca pode ser feito a partir de ações diversas. Aqui, vamos utilizar o percurso em pré-ordem para remover determinado nó da árvore, em vez de apenas retornar o seu valor.

O primeiro passo da remoção é encontrar o elemento que queremos remover. Veja o trecho de código a seguir.

```
private Node removaRecursivamente(Node atual, int valor) {
    if (atual == null) {
        return null;
    }

    if (valor == atual.valor) {
        // o nó a ser removido foi encontrado
        // ... o restante do código para remover virá aqui
    }
    if (valor < atual.valor) {
        atual.filhoEsquerda = removaRecursivamente(atual.filhoEsquerda, valor)
        return atual;
    }
    atual.filhoDireita = removaRecursivamente(atual.filhoDireita, valor);
    return atual;
}
```

Após fazermos a remoção, é preciso garantir que a regra utilizada na inserção dos elementos não seja violada. Para isso, temos que considerar três possíveis cenários: o nó removido é um nó folha, ou seja, não tem filhos; o nó removido tem apenas um filho; ou o nó removido tem dois filhos.

No primeiro caso, é necessário apenas remover o elemento, conforme figura a seguir.

```
if (atual.filhoEsquerda == null && atual.filhoDireita == null)
    return null;
}
```

No segundo caso, temos que retornar o filho do nó removido, pois ele será o novo nó filho.

```
if (atual.filhoDireita == null) {
    return current.left;
}

if (atual.filhoEsquerda == null) {
    return current.right;
}
```

Para tratar o terceiro cenário, quando o nó a ser removido tem dois filhos, vamos considerar que o elemento que assumirá a posição do nó removido é aquele com o menor valor da subárvore da direita do nó a ser removido. O trecho de código na sequência retorna o valor desse elemento.

```
private int encontreMenorValor(No raiz) {
    return raiz.filhoEsquerda == null ? raiz.valor
    encontreMenorValor(raiz.filhoEsquerda);
}
```

Por fim, temos que assinalar o menor valor ao nó corrente.

```
int menorValor = encontreMenorValor(atual.filhoDireita);
atual.valor = menorValor;
atual.filhoDireita = removaRecursivamente(atual.filhoDireita, menorValor
return atual;
```

Assim, conseguimos concluir nossa operação.

A figura na sequência apresenta o código completo da operação de remoção de um nó de uma árvore binária.

```
1  private Node removaRecursivamente(Node atual, int valor) {
2      if (atual == null) {
3          return null;
4      }
5
6      if (valor == atual.valor) {
7         // o nó a ser removido foi encontrado
8         // caso 1
9          if (atual.filhoEsquerda == null && atual.filhoDireita == null) {
10             return null;
11         }
12
13        // caso 2
14         if (atual.filhoDireita == null) {
15             return current.left;
16         }
17
18         if (atual.filhoEsquerda == null) {
19             return current.right;
20         }
21
22         // caso 3
23         int menorValor = encontreMenorValor(atual.filhoDireita);
24         atual.valor = menorValor;
25         atual.filhoDireita = removaRecursivamente(atual.filhoDireita, menorValor);
26         return atual;
27     }
28
29     if (valor < atual.valor) {
30         atual.filhoEsquerda = removaRecursivamente(atual.filhoEsquerda, valor);
31         return atual;
32     }
33     atual.filhoDireita = removaRecursivamente(atual.filhoDireita, valor);
34     return atual;
35 }
36
37 private int encontreMenorValor(No raiz) {
38     return raiz.filhoEsquerda == null ? raiz.valor : encontreMenorValor(raiz.filhoEsquerda
39 }
```

Além desses três métodos, outro conceito essencial para nossos estudos é o de árvore AVL. Vamos conhecer sobre ele e entender suas aplicações no item a seguir. Acompanhe!

4.2.4 Árvore AVL

Uma árvore binária é denominada “AVL” quando, para qualquer nó nela inserido, as alturas de suas duas subárvores (esquerda e direita) diferem em módulo de até uma unidade. O nome AVL vem de seus criadores

soviéticos Adelson Velsky e Landis. Esse tipo de árvore binária tem melhor *performance* nos algoritmos de percurso, justamente pela sua característica de formação (ZIVIANI, 2012).

VOCÊ QUER LER?

Uma leitura interessante sobre as árvores é a apostila escrita pelos professores W. Celes e J. L. Rangel. No arquivo, os autores explicam melhor sobre as árvores binárias e árvores genéricas. Vale a pena acessar o material e se aprofundar melhor na temática. A apostila está disponível na íntegra por meio do *link*: http://www.ic.unicamp.br/~ra069320/PED/MC102/1s2008/Apostilas/Cap13.pdf.

As figuras a seguir apresentam exemplos de árvores AVL.

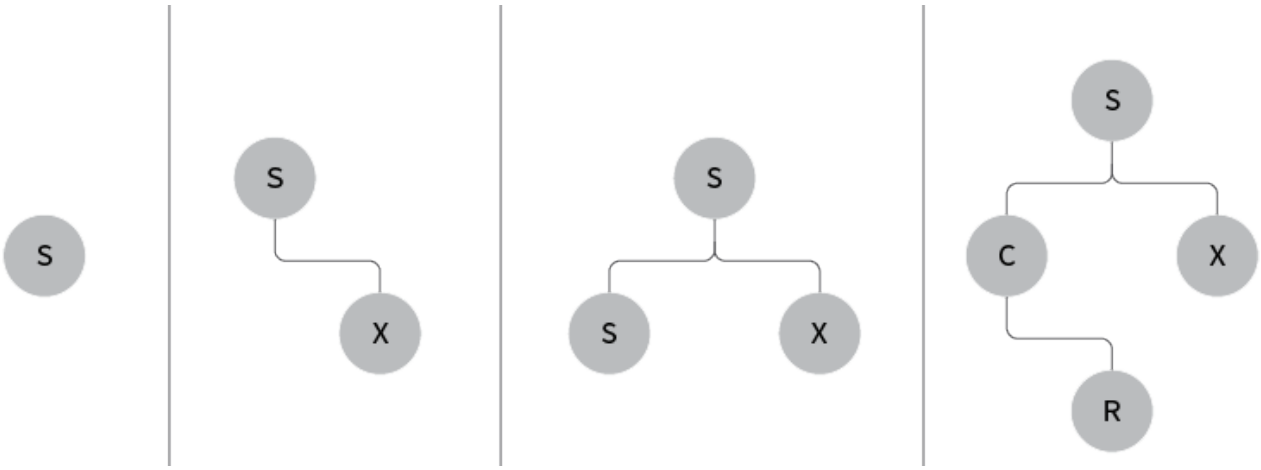

Figura 16 - Exemplos de árvore AVL
Fonte: Elaborada pela autora, 2019.

Veja que, mesmo o nó com apenas a raiz ou um filho, já é uma árvore AVL. Observe, ainda, que a regra relacionada com a altura das subárvores da esquerda e da direita é preservada nesses casos.
Já a figura na sequência nos mostra dois exemplos de árvore binárias que não são árvores AVL. Note que a subárvore da esquerda tem um nível a mais que da direita nos dois casos.

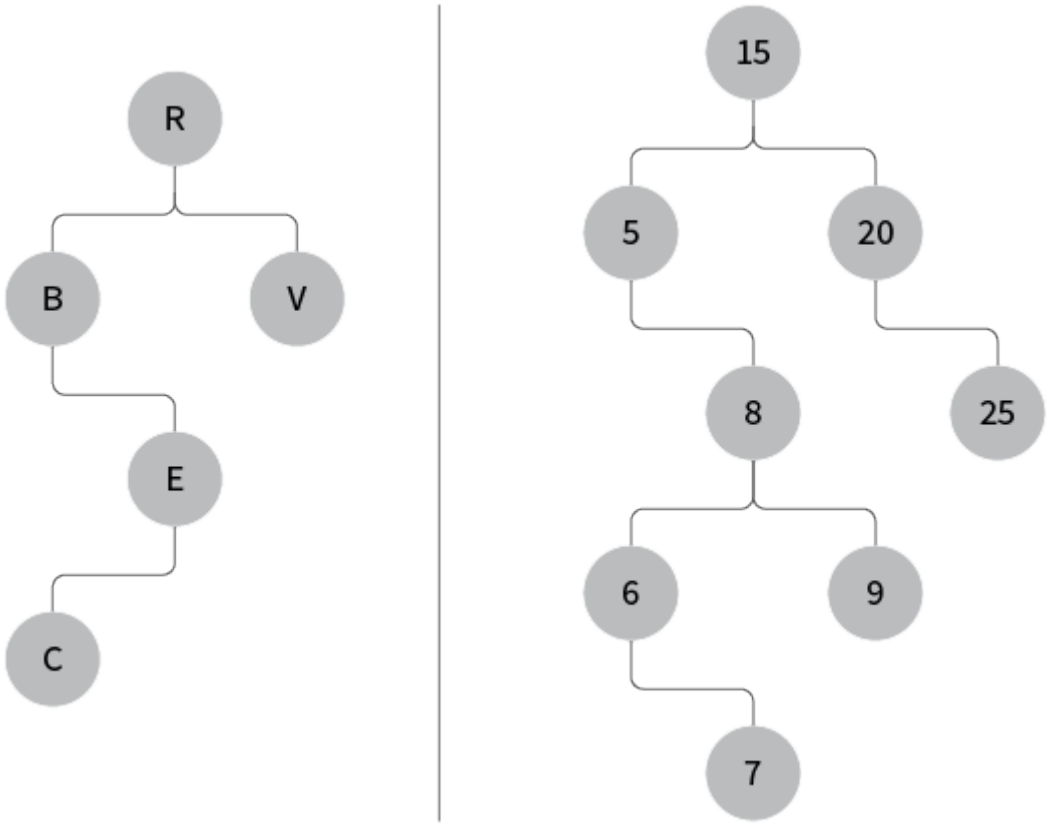

Figura 17 - Exemplos de árvores binárias, mas que não são AVL
Fonte: Elaborada pela autora, 2019.

Para compreender o processo de construção de uma árvore AVL, inicialmente, temos que entender o conceito de árvore perfeitamente balanceada. Segundo Ziviani (2012), a árvore é dita perfeitamente balanceada quando o módulo da diferença da altura das subárvores da esquerda e direita é menor ou igual a 1. Por este motivo, a árvore AVL também é conhecida como árvore binária de busca balanceada. A principal vantagem dessa propriedade é que o tempo médio de pesquisa, em que cada chave é igualmente provável de ser encontrada em uma pesquisa, é menor em comparação com outras árvores binárias.
Encerramos aqui nossa unidade sobre árvores. Assim como as demais estruturas de dados, busque refazer os exemplos, compilar e executar os códigos. A absorção do conteúdo está diretamente ligada à quantidade de exemplos diferentes que você tem acesso.

Síntese

Chegamos ao fim da última unidade da disciplina de Estrutura de Dados. Aqui, estudamos a respeito das árvores binárias e AVL. Vimos que a forma de armazenamento e as operações disponíveis para cada uma delas está diretamente relacionada ao comportamento esperado da aplicação. Além disso, também pudemos aprender um exemplo prático de remoção de nós na árvore binária, mantendo a sua estrutura.
Nesta unidade, você teve a oportunidade de:

- compreender que as estruturas de dados de árvore, com maior enfoque nas árvores binárias, possuem certas características que precisam ser mantidas no processo de inserção e de busca por um elemento;
- entender o passo a passo da construção de uma árvore binária de busca e os três algoritmos principais de percurso nesse tipo de estrutura;
- identificar a diferença entre uma árvore binária e uma árvore AVL;

- compreender que os conceitos, apesar de teóricos, tentam imitar as aplicações práticas do dia a dia;
- criar tipos completos em Java para construir árvores para escopos distintos.

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.

CELES, W.; RANGEL, J. L. **Árvores**. Rio de Janeiro: PUC, 2008. Disponível em: http://www.ic.unicamp.br/~ra069320/PED/MC102/1s2008/Apostilas/Cap13.pdf. Acesso em: 15 ago. 2019.

O JOGO DA IMITAÇÃO. Direção de: Morten Tyldum. Estados Unidos: Diamond Films, 2015. 115 min. Color.

OLIVEIRA, M. Alan Turing: um prodígio além do tempo. **Medium**, 2019. Disponível em: https://medium.com/@mikiasoliveira/um-prod%C3%ADgio-al%C3%A9m-do-tempo-488db8190ba1. Acesso em: 07 ago. 2019.

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

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.