**Tipos de dados primitivos (representação)**

Em Sistemas de Informação, muito se fala sobre dados, informação e conhecimento. Nessa disciplina, trabalha-se, em geral, com os dados. Já a computação, de forma geral, refere-se à informação e ao conhecimento. Previsão ou tendências, por sua vez, são tratadas na Inteligência de Negócios, ou BI (*Business Intelligence*).

**DADOS, INFORMAÇÃO E CONHECIMENTO**

Os **dados** podem ser um conjunto de bits, uma palavra, uma letra, um número, ou podem assumir algum valor sem significado aparente.

Por exemplo: abacate, R$ 5,00, 1 kg.

A **informação** é a união de diversos dados que possuem sentido ou significado.

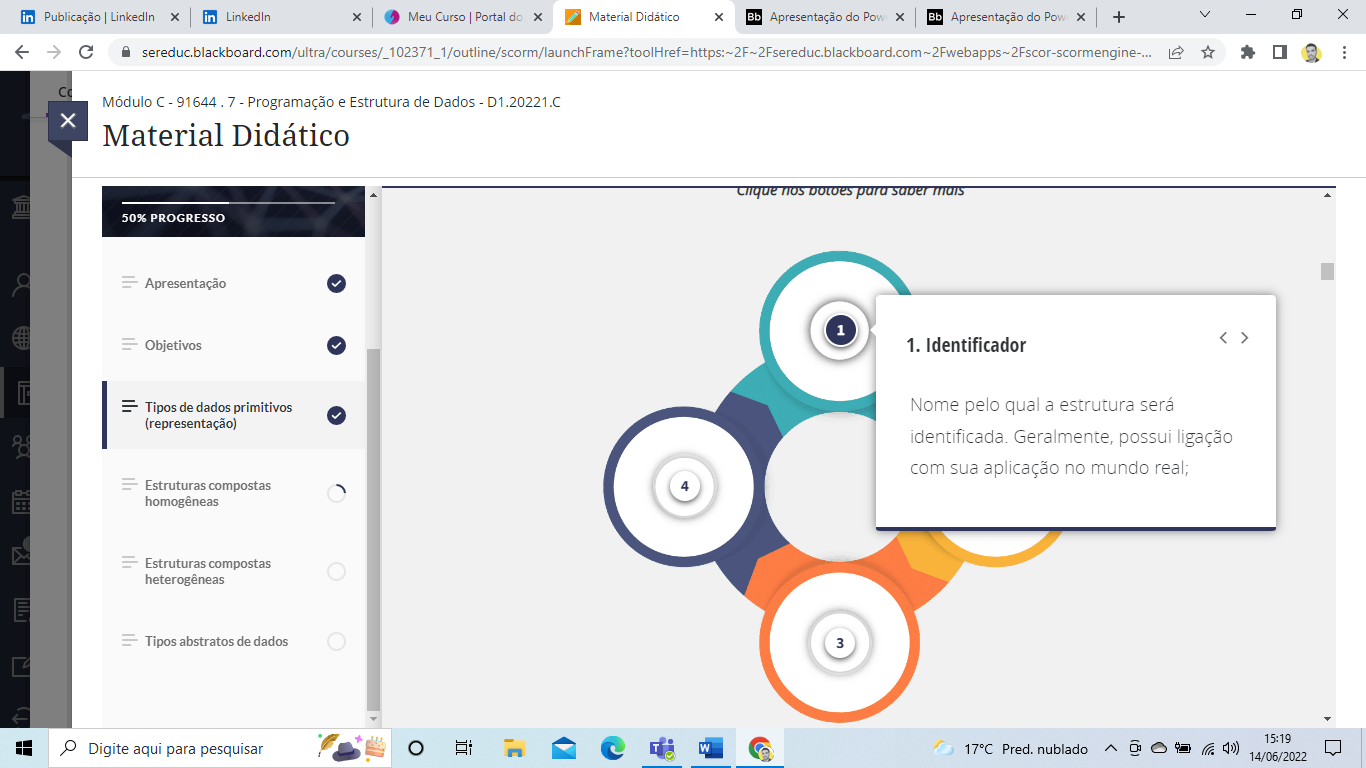
Por exemplo: “Hoje, 1 kg do abacate está custando R$ 5,00 reais”.

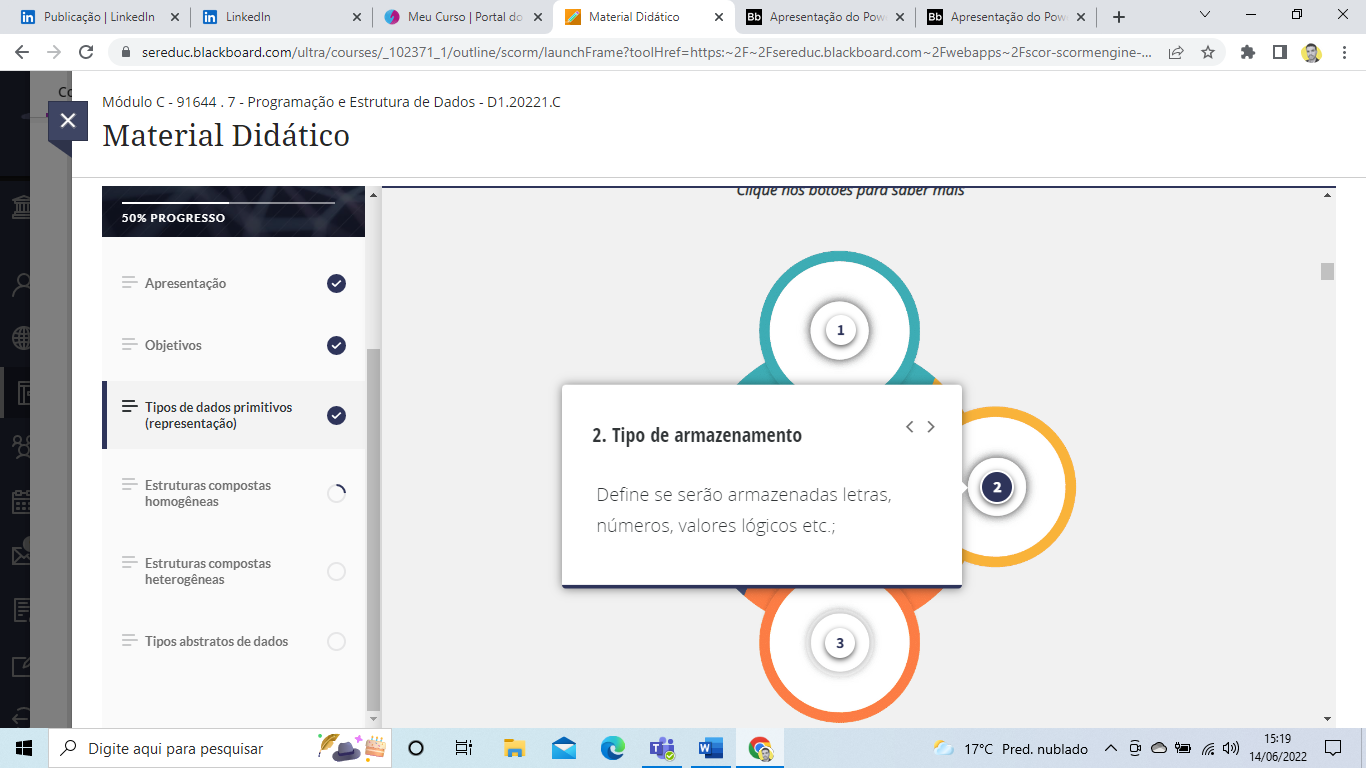
O **conhecimento** é a relação dos dados e informações, constituindo uma ação, aplicação ou saber em diversas áreas do conhecimento, geralmente empregada quando se cruzam valores.

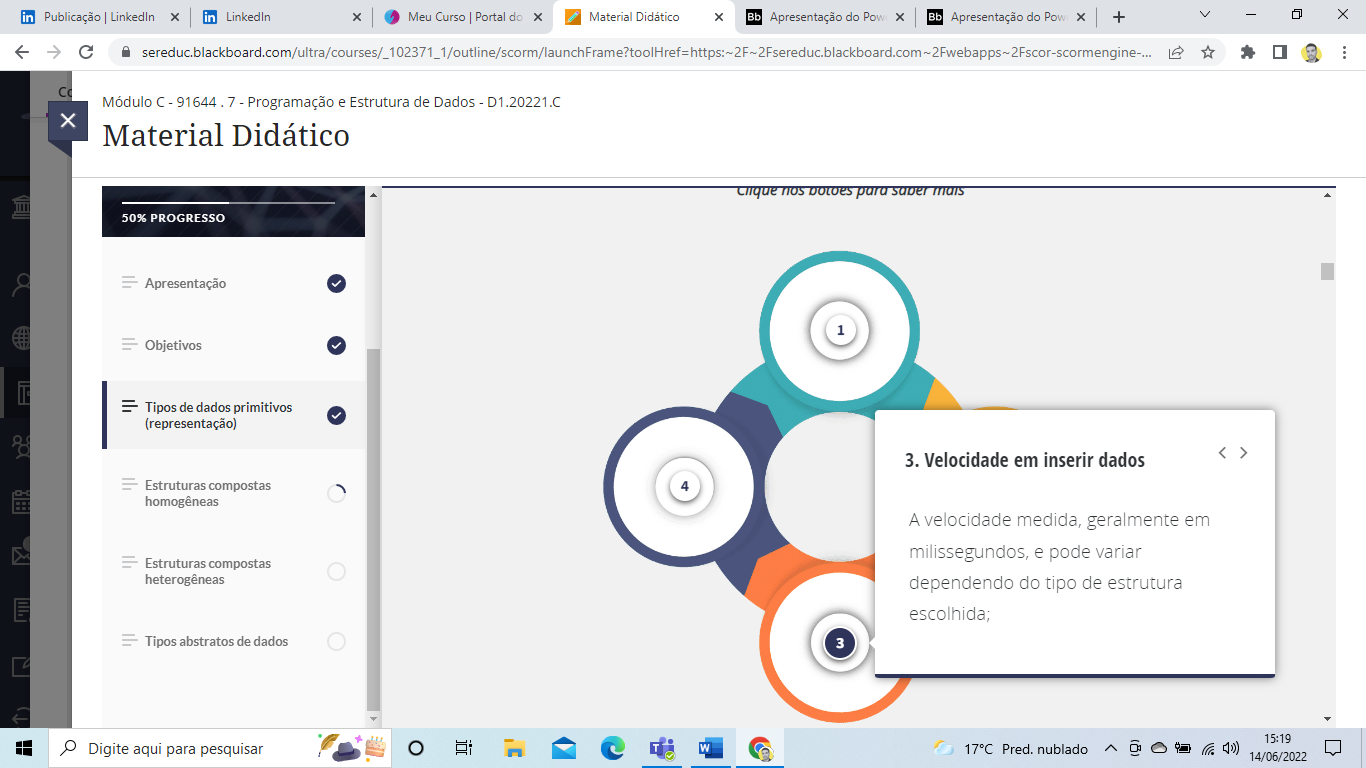
Por exemplo: “Há dez anos, no mês de dezembro, 1 kg do abacate custava R$ 5,00 reais”.

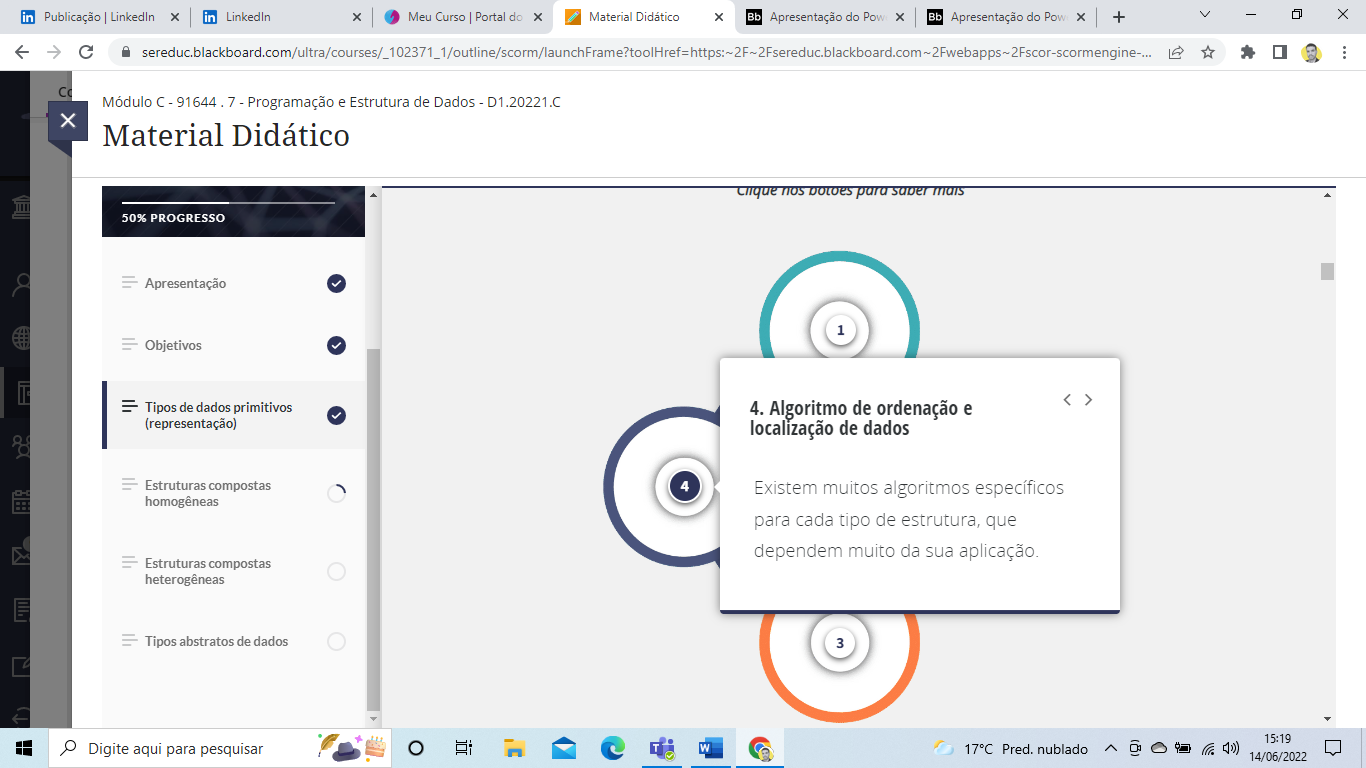
Os **dados**, em Estrutura de Dados, são trabalhados como forma primordial de computar ou solucionar problemas do mundo real, visando atender às **regras de negócio**, ou seja, as regras do mundo real que devem ser incorporadas no mundo computacional, ou atividades empresariais que utilizam esses dados como cadeia de valor do negócio.

A escolha do tipo de dado em Estrutura de Dados leva em consideração alguns aspectos, como:









Às vezes, possuem estruturas que demoram mais para ordenar, mas que, por outro lado, são mais velozes para localizar os dados.

Através do tipo de aplicação de dados, verifica-se qual o melhor formato de dados para tanto. Nesse contexto, surgem os tipos de dados primitivos, que orientam ao programador a estrutura mais adequada para aplicação na regra de negócio.

**TIPOS DE DADOS PRIMITIVOS**

No princípio da comunicação humana, classificávamos nossas trocas de informações em letras, números, sons e escrita. Na tentativa de usar essas classificações da comunicação, na computação, temos os **tipos de dados primitivos**, uma forma do homem (programador) comunicar-se com o computador.

Os tipos de dados primitivos também representam tipos de armazenamento de dados que assumem as seguintes características:

***Clique nos botões para saber mais***

**Cadeia**

–

Para armazenar conjunto de letras, números e caracteres especiais, como: “João”, “três”, “16”, “email@test.com.br”, “(11) 91234-5678”, “Rua A, n° 1 – bairro: B1 – cidade: Cidade 1 – Estado: Es1 – UF: E1”;

**Inteiro**

–

Para armazenar números positivos e negativos sem ponto flutuante, como: 1, -1, 7, 144.000, 40, -88;

**Real**

–

Para armazenar números reais, racionais, tanto positivos quanto negativos, como: 0,5, -0,5, 15,555, número π 3,1415;

**Lógico**

–

Para armazenar valores como verdadeiro e falso, um ou zero, *true* ou *false*. Muito utilizado em toda a computação por ter um bit de espaço, é o tipo primitivo mais veloz no processador, diferente da cadeia que, por conta das operações lógicas executadas para demonstrar uma letra na tela ou no processador, torna-se um dos mais demorados. Porém, existem situações que só se pode utilizar a cadeia, e não outro dado.

Os tipos de dados primitivos são utilizados em algoritmos e em esboços de programação como pseudocódigos, fluxogramas e descrição narrativa.

* O**pseudocódigo** é um esboço em português (ou língua nativa) de instruções de programação.

Por exemplo:

**Algoritmo** Média  
**Declarações**  
n1, n2, média: real

**INICIO**  
           **Leia** (n1)  
           **Leia** (n2)

           Média < -(n1 + n2)/2

**SE** média > = 6  **Então faça**  
           **Escreva** (“Aprovado”)  
       **SENÃO**  
**Escreva** (“Reprovado”)  
         **FIMSE**  
**FIM**

* Os**fluxogramas** são representações de simbólica que representam um algoritmo em forma gráfica.

Por exemplo:

**Diagrama 1**. Exemplo de fluxograma: média aritmética de nota de escola.

**Fonte**: ADALBERTO apud QUINTELA, 2016, p. 7..

* **Descrição narrativa** é o algoritmo dito em língua materna em formato descritivo, ou seja, é como se fosse uma conversa entre dois seres humanos.

Por exemplo:

1) Ler Nota1 e Nota2 do aluno;  
2) Calcular a média aritmética (N1 + N2)/2  
3) Verificar se a média é maior ou igual a 6;  
4) Se sim, aprovado; se não, reprovado.

Os tipos de dados primitivos são, na Estrutura de Dados, o seu pontapé inicial, pois é através dos tipos de dados primitivos que se consegue ter a **abstração**.

**Abstração** é uma técnica da computação e modelagem da matemática formal que consiste em absorver características do mundo real e transformá-lo em computacional.

Por exemplo:

**Hoje** o estudante**Pedro** tirou nota **9,5**de média.  
A abstração dessa afirmação seria retirar o que pode ser computável dessa situação.

**Hoje** – data – **tipo cadeia**;  
**Pedro** – um nome e sua qualificação – **tipo cadeia**;  
**9,5** – um valor numérico "quebrado" como nota de uma média – **tipo real**.

**VARIÁVEIS**

No algoritmo e na computação, temos as **variáveis**, que são definidas como espaço em memória que possui um identificador e pode alocar dados. Como nos tipos de dados primitivos que compõem a cadeia (inteiro, real e lógico), nas linguagens de programação, temos tipos primitivos mais específicos, em que cada linguagem possui seu tipo primitivo, porém, algumas linguagens possuem um padrão.

Nesta disciplina, será utilizada a linguagem Java nos exemplos. Nessa linguagem, para definir uma variável, usa-se o conceito de **declaração de variável**.

Por exemplo:

int idade = 33;

**Int**: é o tipo primitivo da linguagem;  
**Idade**: é o identificador da variável ou nome da variável;  
**=** – símbolo de atribuição;  
**33**: valor a ser atribuído na variável idade;  
**;** – delimitador, ou seja, o final da instrução.

**Char**: valor como letra, caractere ou número, que corresponde à cadeia em algoritmos, porém, tem a característica de receber apenas **um** item, entre letra, caractere ou número, dentro de aspas simples.

Por exemplo:

char letra = ‘a’;  
char caractere = ‘#’;  
char numero = ‘2’;

Não poderia ser escrito, por exemplo, char nome = ‘José’, pois o tipo char não recebe mais que um valor.

***Clique nos botões para saber mais***

**Byte**

–

Valores numéricos sem ponto ou vírgula. Corresponde ao inteiro em algoritmos e pode receber valores entre -128 a +127.

Por exemplo: byte b = 5.

**Short**

–

Valores numéricos sem ponto ou vírgula. Corresponde ao inteiro em algoritmos, e pode receber valores positivos ou negativos, entre -3.2768 a +3.2767..

Por exemplo: short idade = 27.

**Int**

–

Valores numéricos sem ponto ou vírgula. Corresponde ao inteiro em algoritmos e pode receber valores positivos ou negativos, entre -2.147.483.648 a +2.147.483.647.

Por exemplo: int quantidade = 33.000.

**Long**

–

Valores numéricos sem ponto ou vírgula. Corresponde ao inteiro em algoritmos, e pode receber valores positivos ou negativos, entre -9.223.372.036.854.775.808 a +9.223.372.036.854.775.807.

Por exemplo: long renavam = 1.231.231.212.

**Float**

–

Valores com ponto e vírgulas, são números "quebrados" ou flutuantes, sendo obrigatório o uso do f após o número. Corresponde ao real em algoritmo e pode receber valores entre +-140.239.846 x 10 - 45 a +-340.282.347 x 10 + 38.

Por exemplo: float altura = 1.73**f**.

**Double**

–

Como se fosse um float com maior espaço em memória, podendo receber valores entre +-494.065.645.841.246.544 x 10 - 324 a +-179.769.313.486.231.570 x 10 + 380.

Por exemplo: double pi = 3.1415**f**.

**Boolean**

–

Apenas *true* ou *false*, verdadeiro ou falso, corresponde ao lógico de algoritmos, recebendo valores de zero a um, sendo o menor tipo primitivo e utilizado em expressões **booleanas**.

Por exemplo: boolean ligado =*true*.

**EXPLICANDO**

Expressões booleanas são expressões que resultam em *true* ou *false*, independentemente do número de variáveis envolvidas.

Para saber mais, acesse nas referências *O que são expressões booleanas*.

***Clique no botão para saber mais***

**String**

–

Valor com letras, caracteres e números. Corresponde à cadeia em algoritmos, porém com maior capacidade que o char, e seus valores são inseridos por duplas. O string, embora seja nativo do Java, trata de um objeto que recebe um conjunto de char dentro de um mesmo objeto. Não é considerado, necessariamente, um tipo primitivo ou variável.

Por exemplo:

String nome = “José da Silva”;

String telefone = “(011) 91111-1111”;

String endereço = “Rua Uma, N° 1”.

**LINGUAGEM JAVA**

A linguagem Java é uma das principais linguagens do mercado, como mostra o Gráfico 1. O Java tem características de multiplataforma e *open source*, segundo a Oracle, sua detentora.

Java é a base de praticamente todos os tipos de aplicativos em rede, sendo o padrão global para o desenvolvimento e fornecimento de aplicativos incorporados para jogos, conteúdo na web e software corporativo. Usado por mais de 12 milhões de desenvolvedores em todo o mundo, o Java permite desenvolver e implementar aplicativos e serviços incríveis, de maneira eficiente (ORACLE, 2019b).

**Gráfico 1**. TIOBE: ranking mundial do uso das linguagens de programação.

**Fonte**: TIOBE. Acesso em: 25/03/2019.

Para o melhor desenvolvimento da disciplina, faça download do JDK (kit de desenvolvimento) e do Netbeans, cujos links estão nas referências. Primeiramente, faça a instalação do JDK e, em seguida, da IDE (ambiente de desenvolvimento) Netbeans.

**// Executando o Java**

Após a instalação do JDK e o Netbeans, abra o Netbeans. Clique em **Arquivo**, **Novo Projeto**, e aparecerá a tela representada na Figura 1:

**Figura 1**. Ambiente Netbeans: tela inicial de**Novo Projeto**.

Na tela do **Novo Projeto**, clique em **Java**, **Aplicação Java** e **Próximo**, e aparecerá a tela conforme a Figura 2:

**Figura 2**. Ambiente Netbeans: criando **Nova Aplicação do Projeto**.

Nessa etapa, escreva o **Nome do Projeto**(todo exemplo e programa que serão desenvolvidos na disciplina serão um Projeto Novo). A localização física padrão do projeto é na pasta**Documentos\NetBeansProjects** do Usuário local. Deixe selecionado **Criar Classe Principal**.

**Figura 3**. Ambiente Netbeans: tela inicial de **Desenvolvimento**.

Na área de desenvolvimento, temos o botão **Play**, que é o botão que inicia o comando de execução do projeto e o faz na parte de baixo da área de desenvolvimento. Na disciplina, será fornecido o código para execução das estruturas.

Atente-se ao fato de que todo nome do programa deverá (neste primeiro momento) ser o nome do projeto. Logo, para o exemplo a seguir:

**public class prjOla {**  
**public static void main(String []args)**  
**{**  
**System.out.println(“Olá”);**  
**}**  
**}**

O nome do projeto, ao criar um novo, é prjOla. O Java é *case sensitive* (isto é, ele diferencia letras maiúsculas e minúsculas), então prjOLA é diferente de PRJOLA e diferente de prjola. Por isso, nomes de programas, projetos e comandos devem seguir exatamente como estão escritos no material.

No comando anterior, para poder executar, cole o código e digite **play**.

Os comandos em Java funcionam via blocos, que são as aberturas e fechamentos de chaves.

O principal método a ser executado no projeto é o *public static void main*, que, ao ser iniciado, irá apresentar na tela a palavra “Olá” através do comando System.out.println().

A seguir, estão exemplos do uso de tipos primitivos em operação matemática e demonstração em tela.

**public class prjSoma {**  
**public static void main(String []args)**  
**{**  
**int numA = 5;**  
**int numB = 6;**  
**int result = numA + numB;**  
**System.out.println(“O resultado de A + B :” + result);**  
**}**  
**}**

**INSERINDO CLASSES**

Para inserir mais uma classe no projeto, clique com o direito do mouse em cima do pacote, como na Figura 4.

**Figura 4**. Ambiente Netbeans: criando uma nova Classe no Projeto (passo 1).

Aparecerá a janela da Figura 5. Somente escreva o nome da classe e clique em**Finalizar**.

**Figura 5**. Ambiente Netbeans: criando uma nova Classe no Projeto (passo 2).

**Interaja com a dinâmica abaixo:**

**Estruturas compostas homogêneas**

As estruturas compostas homogêneas são aquelas que podem receber diversos valores do mesmo tipo. Os principais representantes dessa estrutura são os vetores e as matrizes.

**VETOR**

O vetor é um tipo de estrutura de dados que é composto por várias instâncias do mesmo tipo e o identificador somente diferencia seu índice, como endereçamento. Ou seja, nada mais é do que um tipo primitivo com profundidade. Por exemplo:

String nome[]=new String[3];  
nome[0]=“José”;  
nome[1]=”Paulo”;  
nome[2]=”João”;

Observe que iniciamos a estrutura do vetor com os colchetes, após o nome da variável. Após isso, usamos o comando **new**, que é um comando que instância o objeto **string**. O valor 3 entre colchetes significa 3 “variáveis” do tipo string, com o identificador **nome[ ]**. Os números entre os colchetes significam sua identificação numérica, que vai até dois. Temos três variáveis com os ids 0,1 e 2, e essa identificação numérica, ou id, são as profundidades do vetor. No caso, o id 0 conta como variável, por isso, vai até o id 2.

**Instância**: convenção da programação orientada a objetos, que cria instâncias ou cópias de objetos ou programas dentro do programa atual.

**Atribuição**: a atribuição é uma característica das variáveis. Elas recebem valores predeterminados pelo programador ou por intermédio de funções que captam valores dos usuários do programa. Abaixo, segue um exemplo de atribuição em vetor.

* Atribuição por valores predeterminados pelo programador:

nome[0]=“Rolfi”;

* Atribuição por valores que o usuário irá inserir:

nome[1]=javax.swing.JOptionPane.showMessageDialog(null, “Digite seu nome”);

* Atribuição por variáveis ou por outro vetor:

nome[0]=outronome;  
nome[0]=nome[1];

**Operação aritmética**: conjunto de instruções que promovem operações aritméticas em variáveis. Por exemplo:

float nota[]=new float[4];  
float media;

nota[0]=7.5f;  
nota[1]=10.0f;  
nota[2]=8.0f;  
nota[3]=10.0f;

media = (nota[0] + nota[1] + nota[2] + nota[3])/4;

**Navegação**: para navegar no vetor, pode-se chamar pelo id ou fazer um *loop* navegando por todos os ids. Nos comandos acima, o vetor **nomes**está sendo impresso, um a um, na tela.

* Navegando pelo id:

System.out.println(nome[0]);

* Navegando em todo o vetor:

for(int contador=0; contador < nome.length; contador++)  
{  
System.out.println(nome[contador]);  
}

Também pelo comando abaixo que é o **for-each**, um dos principais comandos de navegação de objetos e estruturas de dados:

for(int aux: vetor)  
{  
              System.out.println(aux);

}

Esse *loop* navega nas posições do vetor e passa seus valores, um a um, na variável aux.

**Ordenação**: conjunto de instruções que visa a ordenar uma estrutura de dados, nesse caso, o vetor. Abaixo, segue um dos formatos mais tradicionais de ordenação de vetor.

1    for(int contadorL=0; contadorL< vetor.length ; contadorL++)  
2    {  
3           for(int contadorC=contadorL+1; contadorC< vetor.length; contadorC++)  
4           {  
5                 if(vetor[contadorL]> vetor[contadorC])  
6               {  
7                 int aux= vetor[contadorL];  
8                 vetor[contadorL]=vetor[contadorC];  
9                 vetor[contadorC]=aux;  
10            }  
11         }  
12    }

Esse algoritmo é o Método Bolha, ou *BoobleSort*, um algoritmo de ordenação tradicional que busca o menor elemento da lista, posicionando ao topo de forma ordenada. Primeiramente, busca-se o menor valor, navegando no vetor através dos índices contadorL e contador. Ao encontrar um valor maior, faz-se a troca, “descendo” o valor maior e “subindo” o valor menor.

Por exemplo: valores 9, 0, -1 para os endereços 0,1 e 2 do vetor.

**Tabela 1**. Exemplo dos valores iniciais vetor em memória.

Primeiramente, o algoritmo verifica se vetor na posição 0 é maior que vetor na posição 1, ou seja, se 9 > 0. Nesse caso, é maior, então troca-se o menor pelo maior, através das navegações dos índices. Faz-se a primeira troca de 9 por 0, usando a variável auxiliar, como nas linhas abaixo:

7                 int aux=vetor[contadorL];  
8                  vetor[contadorL]=vetor[contadorC];  
9                  vetor[contadorC]=aux;

O aux recebe o maior valor, que é o valor 9, pois o contadorL nesse primeiro momento é 0, e o vetor na posição 0 recebe o valor de vetor na posição 1, que é o valor de contadorC, recebendo, assim, o valor 0. O vetor na posição 1 recebe o valor de aux.

**Tabela 2**. Exemplo da primeira troca no vetor.

Continuando o *loop*, o contadorL na posição 0 verifica se existe um outro valor menor dentro do vetor, encontra na posição 2 o valor –1, e faz a troca pelo valor menor. Logo,todos os valores do vetor são percorridos, como na Tabela 3.

**Tabela 3**. Exemplo da segunda troca no vetor.

Após o primeiro item (o índice 0) possuir o menor valor do vetor, a segunda etapa é passar para verificar se o índice 1 possui o menor valor. Na primeira varredura entre índice 1 com o valor de 9, verifica-se se o próximo índice, o 2, é maior com o valor de 0. Nesse caso, faz-se a troca, obtendo-se o resultado exibido na Tabela 4.

**Tabela 4**. Exemplo do vetor ordenado.

Como não existem valores para serem verificados, finaliza-se a ordenação do vetor.

**Aplicação**: um vetor pode ser usado como lista ou estruturas que envolvam listas, como lista de compras, lista de clientes, lista de CEPs etc.

**// Exemplo do código do vetor estruturado**

No código a seguir, temos um exemplo de vetor em formato estruturado, ou seja, digitado diretamente da função **main**. Esse comando cria um vetor na linha 9 com dados predeterminados e na linha 11 ele ordena esses valores de forma crescente. Após a ordenação, tem-se a impressão na linha 20.

**Figura 6**. Exemplo de vetor em formato estruturado.

**// Vetor orientado a objetos**

Os vetores orientados a objetos são vetores que, tradicionalmente, podem ser criados em qualquer local da classe. Porém, quando abordamos o paradigma de orientação a objetos dentro do conceito de vetor, temos que criar duas classes, uma será o tads propriamente dito, a classe vetorOO, que é chamado como estrutura de dados dentro da classe Prj\_VetorOO.

Empregando os conceitos de orientação a objetos dentro da classe VetorOO, observamos o tratamento de dados através do uso de métodos ou funções, diferente do exemplo acima, em que todo o tratamento está dentro da função main.

**Figura 7**. Exemplo de tratamento de dados através do uso de métodos ou funções.

A classe Prj\_VetorOO possui a **instanciação** da classe VetorOO em vet e, dentro da própria vet, temos a função set, valor que insere valores nas posições determinadas da função.

Também para inserir valores, usamos o **inserir** que, através de uma sequência estipulada na linha 39, apenas recebe os dados e verifica, de acordo com o número de inserções, se poderá ser inserido.

O **construtor** da linha 26 inicia-se recebendo um valor para aprofundar o vetor no valor desejado.

As funções **getValor** e **setValor** são funções de captura e envio de dados em qualquer estrutura.

O **getSize** retorna a quantidade de itens inseridos, enquanto a **getMax** retorna a capacidade máxima de itens que podem ser inseridos.

O **remover** retira itens do vetor diminuindo o getSize e o método **sort** ordena os itens do vetor.

Quando tratamos de orientação a objetos, nesse caso, temos todos os comportamentos dos dados dentro de um vetor, que, no caso, seria a linha 5 private int vetor, que cria o vetor como *private* para que nenhuma classe externa possa acessá-la diretamente, somente a partir dos métodos citados anteriormente.

**Figura 8**. Exemplo de comportamentos dos dados dentro de um vetor.

**MATRIZ**

Matriz é uma estrutura homogênea de dados que permite organizar os dados em formatos de linhas e colunas, e sua indexação é pelo endereço de linhas e colunas, semelhante à estrutura de um vetor, mas com formato bidimensional, podendo assumir mais de duas dimensões. Dependendo da linguagem de programação podem assumir até 17 dimensões, por exemplo:

String nomes[][]=new String[2][3];  
nomes[0][0]=”Rolfi”;  
nomes[0][1]=”Ana”;  
nomes[0][2]=”Maria”;  
nomes[1][0]=”João”;  
nomes[1][1]=”Pedro”;  
nomes[1][2]=”Paula”;

Em estrutura de memória, ficaria ilustrado da seguinte forma:

**Tabela 5**. Exemplo em memória de matriz.

Observe que as colunas são de, no máximo, três itens. Conta-se o 0 e temos 0, 1, 2, totalizando três colunas. Nas linhas, temos dois de tamanho. Conta-se o 0 e temos 0 e 1, como no vetor.

**Atribuição**: as atribuições em matrizes funcionam da seguinte forma:  
nomes[índice da linha][índice da coluna]=valor do tipo da coluna.

Por exemplo:

nomes[0][0]=”Rolfi”;

**// Operação aritmética com matriz**

As operações aritméticas, assim como nas variáveis e vetores, também funcionam da mesma forma na matriz. Deve-se primeiramente fazer referência da posição que se deseja trabalhar e submetê-la às operações matemáticas.

Por exemplo:

float notas[][]=new float[2][2];  
notas[0][0]=10f;  
notas[0][1]=5.5f;  
notas[1][0]=6.0f;  
notas[0][1]=8.0f;

float medias = (notas[0][0] + notas[0][1] + notas[1][0] + notas[1][1])/notas.length;

**Navegação**: para navegar na matriz, pode-se chamar pelo id da linha e coluna ou fazer um *loop*, navegando por todos os ids. Nesse comando, a matriz nomes está sendo impressa, um a um, na tela.

* Navegando pelo id:

System.out.println(nomes[0][0]);

* Navegando em toda a matriz, porém, deve ser feito um laço para a linha e coluna:

for(int contadorL=0; contadorL < 2; contadorL++)  
{  
for(int contadorC=0; contadorC < 3; contadorC++)  
{

System.out.println( nomes[ contadorL ] [ contadorC ] );

}  
}

Ou pelo comando abaixo, que é o **for-each** (um dos principais comandos de navegação de objetos e estruturas de dados):

for(String linha[]: nomes)  
{  
     for(String coluna: linha)  
    {  
         System.out.println(coluna);  
}  
}

Esse *loop* navega nas posições da matriz, passando seus valores, um a um, na variável coluna, que recebe, da variável linha, a linha com os valores da matriz nomes. Para cada dimensão, deve-se ter um *loop*. Nesse caso, temos duas dimensões, então, usa-se dois *loops*.

**Aplicação**: uma matriz pode ser usada como lista em forma de colunas e linhas, com agrupamento de dados semelhantes multidimensionais. Por exemplo, jogos 2D, jogos 3D e muitos jogos antigos eram produzidos apenas em matrizes 2D simples. Com o aumento do processamento, usou-se a 3D, e alguns jogos incluem até 4D, que consiste na dissipação do som através do tempo versus espaço. Também pode-se armazenar os objetos em tela; por exemplo, um cadastro de clientes, compras ou dados organizados em forma de colunas, linhas ou cubo, como em BI.

**Estruturas compostas heterogêneas**

As estruturas heterogêneas consistem em estruturas que podem receber diversos tipos primitivos com um objetivo em comum.

Nesse aspecto, temos os conceitos de registro e campo. O campo é um tipo primitivo com identificador que possui uma característica única que, quando linkada a outros campos, produzem um significado ou conjunto de informações. O registro é uma informação de um determinado objeto ou entidade que possui diversos campos.

Na maioria das literaturas, usa-se o C para explicar estrutura heterogêneas, pois, por não ser orientado a objetos, consegue-se distinguir nitidamente a estrutura heterogênea. Temos o seguinte código:

struct Estudante {  
int ID;  
char nome[50];  
int idade;  
float media;  
};

Em que cada **estudante**adicionado seria um **registro**em memória.

**INSTANCIAÇÃO**

Tendo-se o C, uma *struct*ou estrutura, são estruturas de dados heterogêneos, que possuem diversos atributos (variáveis primitivas) com identificador e tipo primitivo. Logo, sua instanciação, ou cópia, é como no exemplo abaixo.

struct Estudante est1;

**ATRIBUIÇÃO**

Em *structs*, temos dois tipos de atribuição.

* *struct*Estudante est1 = {1,"João",23,10};

ou

* *struct*Estudante est2;

                    est2.ID =2;  
                    strcpy(est2.nome , "Pedro");  
                    est2.idade=30;  
                    est2.media=7.5;

As operações, impressões e acessos são através do ponto, como no exemplo anterior.

Em Java, temos diversas formas de trabalhar uma estrutura heterogênea que difere do C. Uma estrutura consiste em uma classe ou objeto. Um objeto é a principal entidade de uma estrutura de programação, orientada a objetos que possuem propriedades e ações e podem assumir características de uma situação real, uma regra de negócio ou entidade em geral.

    public class Estudante {  
                int ID;  
    String nome;  
    int idade;  
    float media;  
    }

    public class Exemplo {  
      public static void main(String []args)  
      {  
                 Estudante est1=new Estudante();  
   est1.ID = 1;  
}  
    }

Observe que, no exemplo acima, o código está muito próximo da *struct*. No caso do Java também funciona, porém, nos conceitos de orientação a objetos, está fora de padrão e convenção.

No exemplo abaixo, segue o código no formato ideal para orientação a objeto, pois usa-se o conceito de **encapsulamento**, que é o uso de funções *sets*e *gets*para gerenciar as variáveis ou atributos da estrutura. Para alterar valores, usa-se o *set*e o nome da variável.  
Por exemplo:  
    private int idade;

    public void setIdade(int aux) {  
      this.idade = aux;  
    }

Toda variável, ou tipo primitivo, dentro de um objeto que será aproveitada em outras classes devem ser ***private***. Caso não tenha uma palavra reservada como *public*, o compilador entende como *public* por *default*.

Observe que, nesse momento, o nome da função assumiu o nome da variável, acrescentando a palavra *set* como convenção de alteração.

Para captar valores, usa-se o *get*, como na função a seguir:

    public int getIdade(){  
    return this.idade;  
    }

No exemplo a seguir, temos a classe **exemplo**modificada para a classe **estudante**dentro do padrão Java.

Cada uma das variáveis assumiu as propriedades do encapsulamento, tornando-se *privates* e com os métodos ***gets***e ***sets***para sua manipulação.

**Figura 9**. Classe **exemplo**modificada para a classe **estudante**dentro do padrão Java.

Abaixo, a classe **exemplo**seria o formato em que a instanciação está correta dentro dos padrões OO. Na linha 6, atribui-se o valor 1 para a variável ID, através do método setID; na linha 8, verifica-se o valor atribuído no ID através do método getID.

**Figura 10**. Classe **exemplo**.

**Instanciação**: diferentemente do C, o Java é orientado a objetos. Por isso, tudo que é desenvolvido em Java e que tenha interação com dados primitivos, geralmente, é um objeto.

No código abaixo, tem-se a classe **estudante**, que é instanciada através da palavra *new*:

    Estudante est=new Estudante();

**Tipos abstratos de dados**

Os Tipos Abstratos de Dados (TADs) são estruturas que correspondem ao funcionamento de estruturas do mundo real, como pilha, fila, lista, árvore, grafos ou qualquer estrutura que organize, de forma singular, uma coleção de dados. De acordo com Lafore:

O significado de Tipo Abstrato de Dados é mais estendido quando aplicado a estruturas de dados como pilhas e filas. Como em qualquer classe, significa que os dados e as operações podem ser executados neles, mas neste contexto até os fundamentos de como os dados são armazenados tornam-se invisíveis para o usuário (2004, p. 187).

Em algumas situações, podem ser confundidas com estruturas heterogêneas. Porém, o objetivo da TADs é possuir em si uma coleção de dados ou objetos e se responsabilizar na sua organização e tratamento, como: inserção, deleção, ordenação, busca e endereçamento. Um TADs em si gerencia sua própria coleção.

Em termos de códigos, em Java, ou qualquer linguagem de programação, não envolve comandos complexos, e sim uma lógica bem definida que faz com que os dados sejam organizados dependendo de sua estrutura.

***Clique nas abas para saber mais***

**Lista**

–

Comporta-se como uma lista do mundo real, que recebe valores e remove em sequência ou por posições;

**Pilha**

–

Comporta-se como uma organização de pilha, como em pilha de livros, em que o primeiro valor que se insere é o último a ser removido;

**Fila**

–

Comporta-se como uma organização de fila, como uma fila de banco, em que o primeiro valor que entra é o primeiro valor que será removido.

**TIPOS DE ARMAZENAMENTO**

Os tipos de armazenamento em TADs podem variar entre dinâmico e estático. O estático possui capacidade limitada, ou estática, de posições. A dinâmica possui capacidade de alocação por demanda, ou seja, quando existe a necessidade de inserir algum valor, ele adapta-se a uma nova quantidade de valores.

Na capacidade estática, tem-se a vantagem de saber a quantidade de valores. Porém, não podendo ser alterada sua capacidade, faz com que seja um formato de armazenamento muito específico, dependendo da regra.

Na capacidade dinâmica de armazenamento, a vantagem de alocação em tempo de execução faz com que se aumente o leque de uso. Porém, precisa-se de algoritmos diferenciados para ligar os dados. Nesse formato, esse tipo de armazenamento se torna mais complexo que o estático.

**COLEÇÕES**

Os TADs são muito utilizados em diversas linguagens, porém, no Java, existem algumas bibliotecas prontas para as principais TADs, dentro de orientação a objetos. As bibliotecas se chamam **pacotes**.

O pacote que contém algumas das principais TADs é o java.util, que possui o *collection frameworks*, classes e interfaces que possuem, prontas, as TADs que não necessariamente precisariam ser reescritas, apenas instanciadas para uso. As *collections* são divididas em conjuntos, listas e mapas.

**Figura 11**. As interfaces da *collection*. **Fonte**: ORACLE, 2019c. Acesso em: 25/03/2019.

As **listas** trabalham em formatos sequenciais de inserção e remoção. Por serem sequenciais, a busca de dados torna-se mais lenta.

Os **conjuntos** trabalham em formatos algorítmicos para inserção, tornando mais lenta a inserção. A busca, a navegação e a remoção porém, tornam-se mais velozes.

Os **mapas** trabalham organizando os dados juntamente com seus índices, tornando mais complexa a programação. Porém, sua busca é veloz.

Na Figura 12, observa-se as classes que podem ser instanciadas para estrutura de dados das *collections*, dos grandes eixos *list* e *queue* ("sequência") e *set* ("conjunto").

**Figura 12**. Classes *collections*. **Fonte**: JAVA4ALL. Acesso em: 25/03/2019.

**EXEMPLO DO USO DA *COLLECTION***

O programa Prj\_Collections consiste em criar uma *collection*, que receba 100 valores, imprimir em tela os 100 valores, remover um item e substituir um item, sempre chamando a função **imprimir** para mostrar aos usuários. Na linha 9, instanciamos a interface *collection* com o tipo <Integer>, ou seja, inteiro com a identificação col. Na linha 11, podemos utilizar qualquer uma das *collections* da API, como: *set*, *HashSet*,*TreeSet*,*List*, *ArrayList*,*Vector*,*LinkedList*, *Stack*e, para mapas,*Map*,*HashMap*,*Hashtable*e *TreeMap*. No caso, utilizamos o *LinkedList*, que é um formato de lista que faz com que o objeto col assuma o comportamento de uma lista.

**Figura 13**. Exemplo do uso da *collection*.

**Algoritmos recursivos**

Alguns problemas exigem que sejam trabalhados laços de repetições para executar as mesmas instruções em quantidades estabelecidas ou esperadas. Os algoritmos recursivos são uma espécie de **loop** onde possuem um critério de parada. Porém, o mesmo algoritmo “chama a si mesmo com argumentos tratados”.

Exemplo:

O programa a seguir imprime do 10 ao 1 na tela.

**Figura 1**. Exemplo de algorimos recursivos.

O método da linha 4, chamado **recursão**, é iniciado pelo método **main** que é o primeiro método a ser executado da classe da linha 16. Ao ser iniciado, passa-se o parâmetro 10. Esse algoritmo faz a contagem regressiva de 10 até 1, chamando a si mesmo decrementando -1 da variável **contador**. Caso não tivesse um critério de parada, esta recursão na linha 9 com a condição **contador==0**, a execução se estenderia até o travamento do computador, pois ele chamaria a si mesmo perpetuamente, como se fosse um loop. Porém, dentro dele deve ter um critério de parada geralmente com o comando **return**, como na linha 10.

Resumindo, a **recursão** nada mais é que uma função que chama ela mesma, recebendo um argumento e repassando o mesmo argumento tratado, com um critério de parada dentro da própria função.

**Figura 2**. Exemplo de recursão.

O fatorial é um exemplo clássico de recursão, onde seu critério de parada geralmente vem no início da função e a chamada de si mesmo ou recursão, acontece na mesma linha do **return**, como na linha 7, o resultado da nova chamada é multiplicado ao argumento inicial decrementando um.

**Técnicas de ordenação**

A ordenação é um dos métodos de Estrutura de Dados que visa ordenar dados desordenados. E para isso existem muitas formas, como:

* **selection sort;**
* **insertion sort;**
* **quick sort;**
* **merge sort;**
* **heap sort;**
* **radix sort (LSD);**
* **radix sort (MSD);**
* **std::sort (intro sort);**
* **std::stable\_sort (adaptive merge sort);**
* **shell sort;**
* **bubble sort;**
* **cocktail shaker sort:**
* **gnome sort;**
* **bitonic sort;**
* **bogo sort.**

O que diferencia esses métodos é o tempo para ordenação e a forma de troca dos dados. Alguns usam recursão, outros somente loops ou a mescla entre eles. Não existe uma forma única de resolver um problema, porém sempre deve ser observado se o algoritmo se encaixa nas regras de negócio.

Seguem os mais tradicionais: **bubble sort**, **insertion sort**, **quick sort** e **selection sort**.

**MÉTODO BUBBLE SORT**

O método **Bubble Sort** ou método bolha, tem por objetivo a ordenação dos dados a partir de 2 em 2, trocando sempre o maior valor pelo menor valor e percorrendo o vetor por diversas vezes. O nome deste algorítmo remete à ideia de que os dados se comportam como “bolhas” dentro de um tanque, e, embora seja simples, não é tão eficiente quanto outros algoritmos de ordenação quando os dados estão totalmente desorganizados.

Na classe Bubble Sort, por exemplo, temos a função **main**na linha 24, que inicia um vetor com valores desordenados e envia esse vetor para o método **Bubble Sort**na linha 25.

Ao chamar o método na linha 25, inicia-se a ordenação a partir da linha 2 onde este método possui dois loops: um que percorre todos os índices, e o segundo loop que percorre até o penúltimo. A troca dos dados é feita após a verificação da linha 5 no **if** que verifica se o valor do índice -1 é maior que o valor do índice do atual. Ou seja, a troca é feita de 2 em 2.

Após a ordenação, o programa volta para a função **main**, executando a linha 26, chamando outra função que é a imprimir, e executa-se o bloco da linha 16, fazendo uma impressão **for-each** em tela do vetor ordenado, diferente de uma variável que precisa ser uma variável global. Observe que ao executar o código, não precisamos receber o vetor ordenado, ele ordena-se pelo o endereço de memória. E fazendo apenas a referência dele, conseguimos fazer a impressão do vetor ordenado.

**Figura 3**. Exemplo do Método Bubble Sort.

**MÉTODO INSERTION SORT**

O método **Insertion Sort** é um algoritmo de ordenação que percorre o vetor da esquerda para a direita, trazendo os menores valores para a esquerda, e inicia as trocas a partir do segundo elemento.

Exemplo:

Considere um vetor de 5 posições vetor[]=new [] {7,6,4,2,9};

1º - 7-6-4-2-9 – desordenado.

2º - **6-7**-4-2-9 – a primeira troca do segundo elemento pelo primeiro.

3º - **4**-6-7-2-9 – a terceira troca, empurrando os elementos maiores para a direita e os menores para a esquerda.

4º - **2**-4-6-7-9 – a quarta troca, empurrando os elementos maiores para a direita e os menores para a esquerda, fazendo o 2 ser o primeiro item do vetor.

5º - 2-4-6-7-**9**, no caso do 9, como é um elemento maior que o 7, ele permanece no mesmo local, retornando o vetor ordenado.

Observe no código a seguir que na linha 23 o método **main**inicia um vetor desordenado e passa via parâmetro para a função **insertion sort** na linha 2. O primeiro loop percorre o vetor, e o segundo loop percorre da esquerda para a direita ordenando os valores. Após a ordenação, chama-se o método imprimir para mostrar o vetor ordenado.

**Figura 4**. Exemplo do Método Insertion Sort.

**MÉTODO SELECTION SORT**

O método **Selection Sort** busca no vetor o menor valor entre todos os elementos e o insere na primeira posição após a verificação. Diferente do **Bubble Sort** que faz a troca pelo maior ou menor através do **insert**de forma ordenada para todos os elementos.

**Figura 5**. Exemplo do Método Selection Sort.

**MÉTODO QUICK SORT**

O método **Quick Sort** consiste na aplicação do método **dividir**e **conquistar**aplicando **recursão**. Divide-se o vetor em partes, buscando um valor arbitrário chamado de **pivô**. Os valores ordenados à direita são maiores que o pivô, enquanto os valores à esquerda são menores. Se chamado de forma recursiva, faz a mesma ordenação entre as **subarrays**. Consiste em um algorítmo eficiente e rápido, geralmente usado quando precisa-se de performance.

**Figura 6**. Exemplo do Método Quick Sort.

**Estruturas lineares**

As estruturas lineares são estruturas que agrupam dados em sequência ou em uma ordem pré-determinada. Como por exemplo, a ordenação em Lista sequencial **FIFO**(*First in First Out* – o primeiro que entra é o primeiro que sai) e **LIFO**(*Last in First Out* – o último que entra é o primeiro que sai), porém sua retirada pode ser tanto em sequência como de forma desordenada, por prioridade, etc.

As estruturas lineares mais utilizadas são: **Lista**, **Pilha**e **Fila**que implementam o vetor dentro de suas estruturas e de acordo com seu comportamento. O armazenamento tem sua implementação diferenciada no código.

Quando fala-se em Listas, no geral, implementa-se a estrutura primitiva do vetor, que é uma estrutura sequencial indexado com o mesmo tipo. E quando implementa-se uma estrutura em vetor por herança, absorve a maior característica do vetor à sua limitação quanto ao tamanho.

A limitação quanto ao tamanho do vetor faz com que as estruturas possam ser dividas em duas formas de alocação: a **estática**e a **dinâmica**.

A **alocação estática** faz com que qualquer estrutura tenha um tamanho máximo para inserção de elementos, forçando o programador a usar soluções paliativas para manter a estrutura em vetor.

A **alocação dinâmica** é uma estrutura que não implementa o vetor, porém implementa o conceito de **No**, onde é possível alterar sua estrutura de acordo com a necessidade do programador.

**Listas**

As listas são estruturas que alocam de forma sequencial os dados. Temos as **listas estáticas** que implementam o vetor, e a **lista dinâmica** que são as listas **ligada** e **duplamente ligada**, tratando-se do *Java Collections* da *interface List*.

**LISTA ESTÁTICA**

A Lista Estática implementa o vetor, podendo ser de forma estruturada ou objeto. Tanto uma quanto a outra são fixas a seus tamanhos e requerem soluções paliativas após o tamanho máximo ser utilizado. A grande vantagem da lista estática ou do uso do vetor na estrutura é a capacidade de indexação dos elementos, fazendo com que a individualização de cada um seja mais efetiva. Diferente de outras estruturas que é necessário navegar na estrutura para acessar os valores.

Exemplo:

System.out.println( vetor[11]); // Neste código a partir do índice 11, busca-se o valor do vetor na posição 11

**LISTA LIGADA**

É um tipo de dado abstrato que se comporta como o conceito de Lista, mas implementa o **Objeto No** ou **Node**.

**// No ou Node**

O **No ou Node** é um objeto que faz autorreferência, tendo um formato onde pode ter de 0 a n itens lincados.

No conceito de **No**, os itens podem ser inseridos e removidos em qualquer local da estrutura, podendo aumentá-los e diminui-los conforme a necessidade. Sua navegação ocorre por meio da referência do objeto denominado próximo, acessado através do método **getProximo()**.

O **construtur** na linha 6 é uma função que recebe como parâmetro o valor a ser alocado e a referência do Objeto que será o próximo do Objeto atual.

Na linha 3, armazena-se o próximo item, que para acessá-lo é preciso usar o método **getProximo()**, e, para setar a referência do próximo No, usa-se o método **setProximo()**. E as funções **setValor()** e **getValor()**, armazenam os valores a serem alocados.

Ao tratar-se da **Lista** ou qualquer outra estrutura de dados, deve-se obrigatoriamente implementar o código do No para ser referenciado na estrutura.

**Figura 7**. Implementação do código No.

A Lista Ligada é uma estrutura que implementa **Node**, e cada elemento dela consiste em um **Node** que possui em si mesmo a referência de seu próximo, possibilitando uma capacidade N de armazenar e diminuir itens conforme a necessidade. Possui como principal objeto o **proximo** que armazena o endereço do próximo objeto, e o valor que armazena os valores alocados.

**Figura 8**. Implementação do Node na estrutura de Lista Ligada. (Adaptado).

Observe a seguinte estrutura de Lista Ligada:

**Figura 9**. Estrutura de Lista Ligada. (Adaptado).

O **primeiroNo** é o início do No, trazendo em si a referência do próximo No que seria o que armazena o -1. Para percorrer os nos, usa-se o método **getProximo**(), e no seu retorno usa-se o **getProximo**() do outro **No**. Assim chega-se ao **ultimoNo** que ao usar o **getProximo**() do **ultimoNo**, ele retornará **null**, assim caracterizando que é o último No da estrutura.

**// Inserção**

A inserção dos elementos 5 e 7 nas extremidades ocorre após, primeiramente, os **Nos** com os valores serem criados e depois inseridos nas extremidades. No caso do 5 que está alocado no início, apenas irá setar como seu próximo, o **primeiroNo**, fazendo assim o segundo **No** da estrutura. No exemplo da inserção no último No, temos ele setando o seu objeto próximo para o novo No, tornando o novo **No** o último, e ele oo penúltimo **No** da lista.

**Figura 10**. Inserção dos elementos 5 e 7 nas extremidades. (Adaptado).

**// Busca de um elemento**

A busca em uma estrutura de Lista Ligada se dá pela navegação da função getProximo(), que sempre irá fornecer o endereço do próximo **No**, e ao chegar no final ele irá retornar null, assim o algoritmo faz um loop entre os getProximo() e confere o valor do no aux com o valor pesquisado, retornando true se localizou ou não o item na lista.

**Figura 11**. Busca de um elemento em uma estrutura de Lista Ligada. (Adaptado).

**// Remoção de um elemento**

Para remover um elemento no No, usa-se o mesmo conceito da busca, um *loop* para achar o No que contém o valor a ser removido. Ao achar o No, e criar um objeto temporário chamado **atual**, é necessário fazer uma busca pelo seu No anterior. Ao localizar, armazená-lo em **aux** e fazer a referência de **aux** na função **setProximo()** no valor do próximo de **atual**. Sendo assim, o **No** que armazena o 0 é lincado ao **No** que armazena o 10, e automaticamente o No que armazena o -1 é retirado da Lista.

**Figura 12**. Remoção de um elemento no No. (Adaptado).

**CÓDIGO LISTA LIGADA**

O código para a Lista Ligada, deve primeiramente, inserir a classe **No** para ser localizada. Como temos neste material a **Lista Ligada** e **Lista Duplamente Ligada**, para não confundir, usou-se neste exemplo o **No**, e o **No** da **Lista Duplamente Ligada** será **Node**, pois possuem atributos parecidos. Os códigos desta classe foram explicados no início deste trabalho.

**Figura 13**. Código da Lista Ligada.

A classe **Lista Ligada** instância o No nas linhas 3 e 4, onde o primeiro **No** e último **No**, armazenam as extremidades da estrutura da lista.

No **construtor** da linha 7, temos os endereços dos **Nos**. Como é o início, eles estão em posição **null**.

A função ***isEmpty*** retorna se o primeiro **No** é nulo ou não. Sendo nulo, significa que não possui **Nos** na estrutura.

**Figura 14**. Funções da Lista Ligada.

As funções **adicionaInicio** e **adicionaFim**, são funções que recebem valores e adicionam os **Nos** recém-criados nas extremidades da estrutura. No caso do **adicionaInicio**, ela primeiramente verifica se a estrutura possui **No** na estrutura, se sim, ela adiciona um novo **No**, fazendo endereçamento do **Novo No**, como primeiro item na linha 22.

Caso não tenha nenhum No na linha 20, cria-se um novo No, setando como **ultimoNo** e **primeiroNo** o novo No criado, com referência ao **null**, tornando-o assim o primeiro e ultimo No da estrutura. Assim também segue na linha 30 do **adicionaFim**.

No caso do **adicionaFim**, ele cria um novo **No** caso já tenha um na estrutura, setando o novo No como último, e tornando o **ultimoNo** o penúltimo No, como na linha 32 e 33.

A linha 38 trata do **imprimir**, que faz uma varredura dentro da estrutura navegando através do **getProximo** em todos os Nos e imprimindo o valor do no atual.

**Figura 15**. Função imprimir.

Observe que tanto a **função imprimir** quanto a **buscar** fazem varreduras na estrutura toda através do **getProximo**. No caso do **imprimir**, ele percorre a estrutura, e o **buscar** percorre (linha 54) e verifica se o valor **aux** está contido dentro dos Nos. Se sim ele retorna true, não tendo retorna false.

**Figura 16**. Função buscar.

O método **excluir** recebe o valor que será excluído, e na linha 69 faz um loop até o final da estrutura. Ao percorrer, busca-se o valor através da linha 71, parecido com o buscar. Ao localizar, cria-se um **No auxiliar aux**, que irá receber o No anterior ao localizado, e como na linha 75, seta o **setProximo** de **aux** para o **getProximo** de atual.

Caso o anterior seja o atual, quando aconteceria se fosse o primeiro item a ser excluído, como na linha 78, o primeiro **No** receberia seu **setProximo** com o **getProximo** do atual, ou seja, se excluiria retornando *true*.

**Figura 17**. Método excluir.

O **prj\_ListaLigada** adiciona itens nas linha 6 a 10, adicionando 1,2,3 e 0. Observe que 1 foi adicionado no inicio, 2 e 3 no fim, fazendo a sequência. E após a numeração 1, 2 e 3 o 0 foi adicionado no inicio do 1, fazendo do 0 o primeiro elemento.

O código faz impressão da lista na linha 12, realiza busca do elemento 3 na lista, e remove o número 6 nas linhas 16 e 18 respectivamente, fazendo a impressão do estado final da estrutura de dados **lista ligada**.

**Figura 18**. Código prj\_ListaLigada.

Ao executar os códigos mostrados, aparecerá a seguinte mensagem:

**Figura 19**. Mensagem ao fim do processo.

**LISTA DUPLAMENTE LIGADA**

A Lista Duplamente Ligada é uma evolução da Lista Ligada, pois a mesma apenas armazena o endereço do seu próximo e, caso ultrapasse para o próximo na navegação, não consegue fazer o caminho de volta entre os Nodes.

O armazenamento do endereço se dá nos objetos **próximo** e **anterior** do **Node**, como nas figuras a seguir:

* 1
* 2

**Figura 20**. Armazenamento do endereço. (Adaptado).

**Figura 21**. Exemplo de referenciação. (Adaptado).

**// Inserção de um elemento**

A inserção dos valores 5 e 7 nas extremidades se dá primeiramente pela criação do **Node** com os valores, e, depois, são apontadas suas referências aos elementos da estrutura.

**Figura 22**. Inserção de um elemento. (Adaptado).

**// Remoção de um elemento**

Na primeira estrutura do algoritmo, é percorrido o elemento que busca-se apagar, e armazena-se o endereço do **No** no objeto **atual**. Após localizar o objeto a ser removido, faz-se uma nova varredura, percorrendo o objeto anterior ou atual, e armazena a referência em **aux**. Ao localizar o **aux**, coloca o endereço do próximo **No** do atual em **aux**, assim liga-se **aux** com o próximo do atual e automaticamente o atual sai de memória.

**Figura 23**. Remoção de um elemento. (Adaptado).

**CÓDIGO LISTA DUPLAMENTE LIGADA**

A Lista Duplamente Ligada segue alguns padrões da Lista Ligada. Primeiramente, usa-se a classe **Node**, onde contém as referências anteriores e posteriores do **No** atual.

Na linha 3, o objeto autorreferenciável **proximo** aloca o endereço do próximo **No**, e o objeto autorreferenciável anterior aloca o endereço do **No** anterior para o primeiro **No** de uma estrutura. O item anterior será **null** enquanto o último **No**, e o endereço do **proximo** será **null**.

**Figura 24**. Código da Lista Duplamente Ligada.

O código da estrutura da **ListaDuplamenteLigada** instância dentro dela a classe **Node**, e implementa as funções:

**isEmpty()** – que retorna true se estiver vazia e false para uma estrutura com conteúdo, onde verifica-se pelo objeto **primeiroNode**.

Na linha 16, **adicionaInicio()** recebe como parâmetro o valor a ser inserido na Lista. Primeiro verifica-se se a lista está vazia, caso esteja, o endereço tanto do último como do primeiro **No** são nulos. Caso não seja o primeiro valor a ser alocado, cria-se um novo **Node**, endereçando o Node atual no seu objeto **primeiroNode** na linha 20, fazendo sua alocação para a esquerda.

Na linha **adicionaFim**, faz-se o mesmo processo do **adicionaInicio**, porém somente remete como parâmetro o **ultimoNode** na linha 29, fazendo sua alocação para a direita.

**Figura 25**. Funções da Lista Duplamente Ligada.

Os comandos da classe **Prj\_ListaDuplamenteLigada** adiciona 5 elementos, onde 3 são no início e 2 no fim, nas linhas 8 até a 12. Após alocação, são impressos e mostram o resultado da busca pelo valor 2, e exclui o valor 1, mostrado na saída os respectivos valores.

**Figura 26**. Comandos da classe Prj\_ListaDuplamenteLigada.

Ao executar esses códigos, aparecerá a seguinte mensagem:

**Figura 27**. Mensagem ao fim do processo.

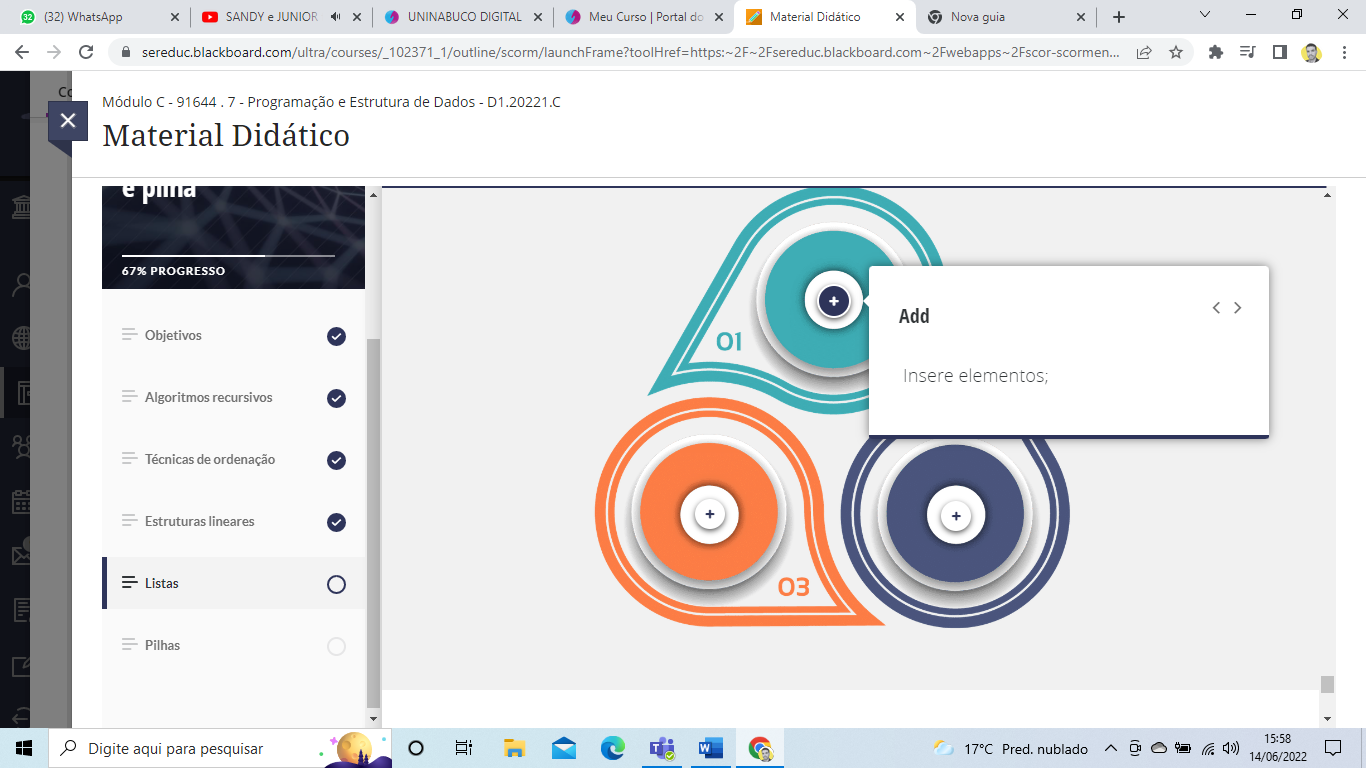
**LISTA COLLECTION**

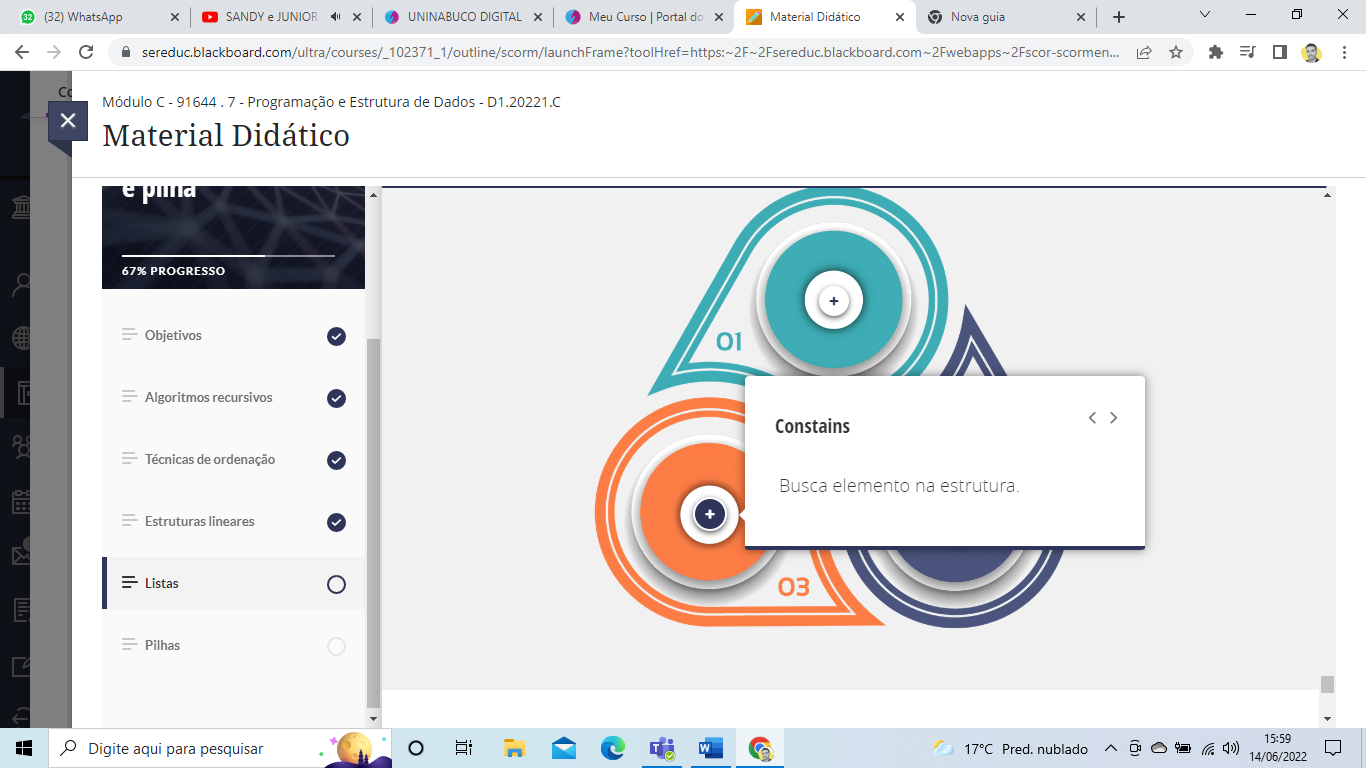
A lista desenvolvida no formato estático implementa a *interface Collection*, no código a seguir (linha 3), importa-se o pacote **util.Collection** que contém as classes que podem ser implementadas da Collection como: *Vector*, *ArrayList* e *LindekList* que fazem parte do formato sequencial ou linear de alocação dinâmica.

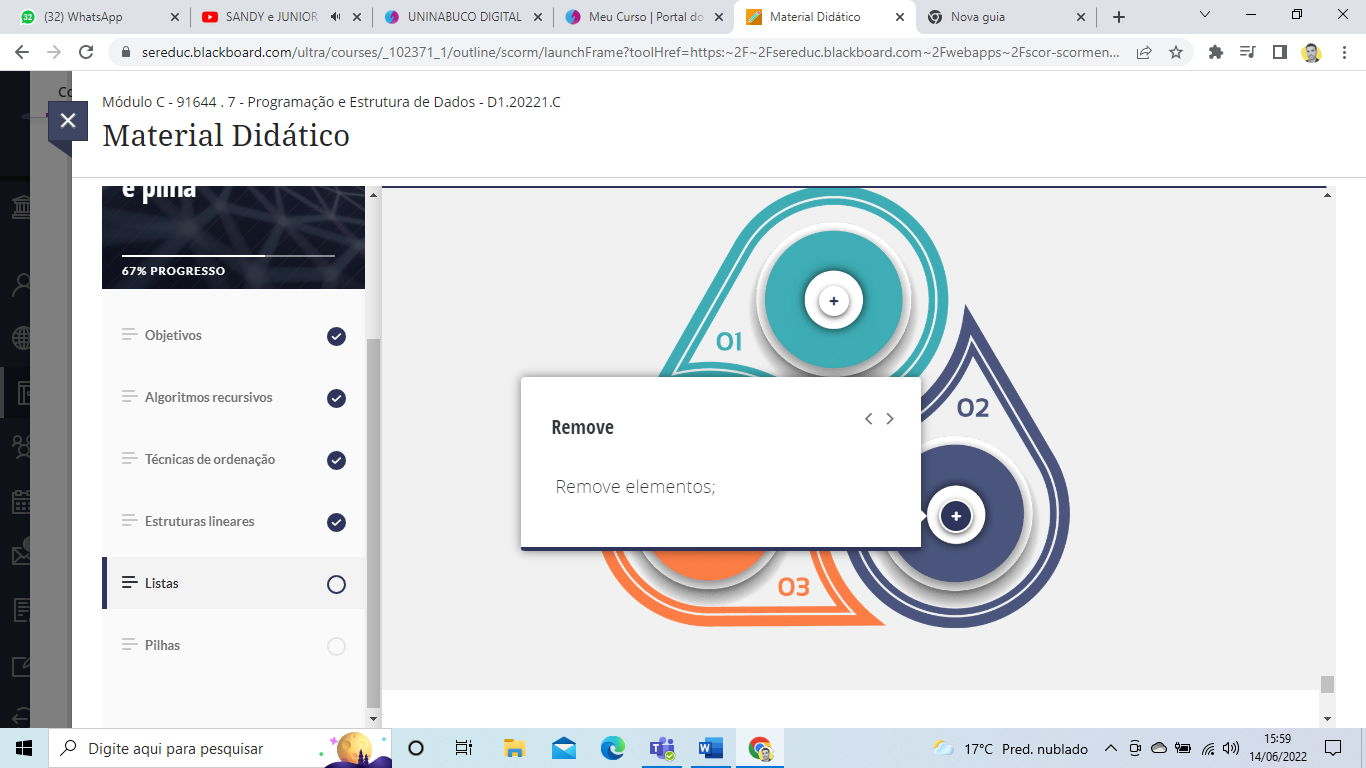
**Figura 28**. Lista Collection.

Na linha 11 utiliza-se o *LinkedList*, na **instanciação** poderia ser usado qualquer Collection sequencial como *Vector* ou *ArrayList*, seguindo o código, usa-se o **<Integer>**. Com esse formato, coloca-se o tipo do primitivo ou objeto que irá ser utilizado na estrutura; neste caso o objeto **col** recebe somente valores do tipo inteiro.

Quando se usa a *interface Collection* as funções são padronizadas independente da estrutura utilizada, como:







No código visto nas linhas 9 e 11, faz-se a **instanciação**. Nas linhas 13 e 14 são inseridos 100 valores na estrutura dentro do loop, e a impressão da estrutura é feita nas linhas 16 e 20. A impressão via for-each na função “imprimir”.

**Pilhas**

A **TAD** (Tipo Abstrato de Dados) **Pilha** é uma variação da **Lista Linear** que possui características semelhantes a uma pilha do mundo real. Ela armazena os valores inseridos no formato de LIFO - *Last in First Out* (O último que entra é o primeiro a sair).

Como uma Pilha de livros, onde o primeiro a ser empilhado é o último a ser acessado, segue um exemplo de empilhamento de dados:

**Diagrama 1**. Exemplo de empilhamento de dados.

A pilha possui as seguintes operações:

* 1

**push( X)**: aloca ou adiciona o valor na estrutura Pilha;

* 2

**pop()**: desempilha ou remove o valor do topo da Pilha, retornando o valor pela função;

* 3

**peek()**: acessa o topo da Pilha sem removê-lo;

* 4

**isEmpty()**: retorna o estado atual da Pilha se possuir itens alocados false, se não possuir retornatrue;

* 5

**isFull()**: retorna true se a Pilha está com todos os espaços alocados, porém para Pilhas que são de alocação dinâmica, não possui a função***isFull()***, somente as Pilhas que implementam o vetor.

**CÓDIGO DA PILHA COM ALOCAÇÃO ESTÁTICA**

**Figura 29**. Código da Pilha com Alocação Estática.

Observa-se na linha 6 o **Construtor** que recebe o tamanho máximo da Pilha. Para poder colocar os itens e a partir daí ele implementa um vetor com esse tamanho máximo e a partir do momento que ele tem tamanho máximo ele tem aquela quantidade de itens para poder colocar dentro do nosso vetor que está sendo implementado na Pilha.

Na linha 11 temos a função **isEmpty** que verifica se o topo = -1. Isso significa que se topo for -1 ela está vazia, então vai retornar**true**. Na linha 15 temos os isFull, ele verifica se a pilha está cheia, ou seja, se o topo é equivalente ao número da quantidade de elementos do **vetorPilha.length** (o atributo que retorna o tamanho máximo do vetor).

O **peek** nada mais é do que o retorno do valor que está no índice topo da pilha na linha 19. O método **push** é a função de adicionar itens dentro da pilha incrementando o topo e atribuindo valor na Pilha no índice topo. E o método **pop** é a função para remover itens da Pilha, fazendo decremento no topo.

A seguir é mostrado o código que usa a **classe Pilha** como estrutura de dados para ser implementada. Neste caso, faz-se a instanciação na linha 4 e denomina essa “cópia” da classe Pilha como pilha em minúsculo. Esta classe adiciona 4 valores e imprime em tela seu topo através da peek o resultado de ***isEmpty*** e ***isFull***. Mas para imprimir todos os itens da Pilha, é necessário removê-lo para apresentá-lo, diferente das estruturas de Lista que possui um índice para mostrar todos os itens.

**Figura 30**. Código que usa a classe Pilha como estrutura de dados para ser implementada.

**COLLECTION CLASSE STACK**

A Classe Stack está dentro da interface Collection, implementa a alocação dinâmica com Node, ela possui diversas funções da embutidas, como por exemplo a push, pop, contains, peek, entre outras. Essas funções que poderiam ser implementadas na mão, porém neste formato já está pronto na Classe Stack, fazendo com que ganhe produtividade ao programar.

No caso da classe Pilha, ela implementa a classe**Stack** onde, no exemplo do código anterior, implementava o vetor. Neste formato, as funções**insere**, **atual**, **ultimo** e **remove** vem como um substituto na classe para chamar as funções pré-programadas como ***push***,***peek***,***firstElement*** e***remove***.

**Figura 31**. Classe Stack.

Na função **limpo** e**pesquisar**, linhas 33 e 45 respectivamente, são utilizadas funções da própria **Stack**como a ***empty()*** que retorna **true** se estiver vazia, e a **search** que busca o elemento dentro da Pilha.

Na linha 37 na função***toString()***, usa-se a convenção da computação para transmitir os valores da estrutura em ***Strings***, no caso com as informações de capacidade usando a função **capactity()** que retorna quantos elementos podem ser inseridos, não obstante quando o tamanho máximo é atingido, aloca-se mais espaço em memória. E a função ***size***que retorna a quantidade de elementos alocados na memória.

**Figura 32**. Função limpo e pesquisar.

O código**Prj\_Pilha** faz uso da classe Pilha como sua estrutura de dados, fazendo a instanciação dela na linha 5, e adicionando elementos na linha 6 até a 12, chamando as funções ***remove***,***limpo***, ***print***, ***pesquisar***, ***atual***,***ultimo*** e ***toString*** respectivamente, mostrando em tela o resultado para esses números inseridos.

**Figura 33**. Código Prj\_Pilha.

**ORDENANDO COLLECTION**

MuitasCollections implementam o**sort**, que é uma função que trata de ordenar os dados de uma coleção através do comando **Collections.sort(nomeestrutura)**; desse modo, as estruturas criadas com Collections não precisam da criação de algoritmos de ordenação.

* Exemplo: ***Collection.sort( vetor)***;

**Filas – implementações**

As Filas são estruturas de dados ou TADs que armazenam no formato **FIFO**.

**EXPLICANDO**

First-in-first-out, ou FIFO, significa que o elemento que é inserido é o primeiro a ser removido, analogamente a uma fila de banco ou qualquer fila do mundo real.

**Tabela 1**. A estrutura Fila.

Os comportamentos padronizados da Fila são:

***Clique nos botões para saber mais***

**Queue(objeto)**

–

Insere o elemento objeto ou valor na Fila;

**deQueue()**

–

Remove os elementos da fila conforme ordem de entrada;

**Peek()**

–

Assim como a Pilha, a Fila trabalha o conceito de topo, que nada mais é do que o elemento em evidência na estrutura. No caso da Fila, é o primeiro elemento que foi inserido, enquanto, na Pilha, o último elemento que foi inserido;

**isEmpty()**

–

É a função que retorna true para uma estrutura vazia e false para uma estrutura que já contém pelo menos o primeiro elemento inserido;

**isFull()**

–

É uma função que retorna booleano, ou seja, true para uma estrutura completa ou cheia e false caso se possa inserir mais elementos, considerando uma Fila de alocação estática que possui tamanho fixo.

O código a seguir demonstra a aplicação do vetor[] no comportamento de uma fila que implementa o formato FIFO. A classe FilaEstática é uma estrutura que dá permissão para outros programas acessarem como objeto uma fila estática (estrutura com tamanho fixo) com tamanho passado via construtor. Na linha 7 tem-se o construtor que inicia a estrutura fila com o tamanho passado pelo programador.

As funções isEmpty e isFull retornam resultados booleanos, respectivamente, se a estrutura está vazia e se a estrutura está cheia.

A função peek retorna a posição do elemento corrente da estrutura, que no caso é sempre a posição 0 do vetor, pois tratando-se da aplicação da FIFO, a posição mais importante da estrutura é o primeiro elemento, como se vê na figura 1.

**Figura 1**. Aplicação do vetor.

A função enQueue ou enfileirar recebe por parâmetro o argumento valor que insere no vetor privado o elemento na posição qtdElementos, variável que controla a posição de quantidades de elementos na fila. Ele se inicia com -1 e, conforme aumenta o número de elementos, incrementa, e, quando remove elementos, decrementa, controlando a quantidade de elementos da fila, o que o torna o principal parâmentro para saber se a fila está cheia ou não.

**Figura 2**. Função enQueue.

A função deQueue ou desenfileirar tem por responsabilidade o comportamento de retirar o elemento do vetor fila e fazer com que todos os elementos avancem uma posição, através do loop da linha 38. Na linha 36, retorna o valor do primeiro elemento e incrementa a quantidade de elementos da fila na linha 37.

A impressão da Fila se dá na função imprimir, que se refere aos valores da linha 47 no loop for-each. A classe Prj\_FilaEstatica aplica a estrutura FilaEstatica e utiliza como estrutura de dados, na linha 5 se faz a instanciação, das linhas 6 a 9 se faz a inserção de elementos na fila e na linha 11 se faz uma remoção e a impressão do valor da fila.

Na inserção, são adicionados os elementos -1, 25, 5 e 7, respectivamente; no caso da organização FIFO, o primeiro elemento se torna o -1 e, ao verificar seu peek ou removê-lo, será trabalhada sempre a primeira posição do vetor, que no caso é 0, que está amarzenado nesse primeiro momento o -1.

Para a remoção, se chama a função deQueue, na linha 11. Primeiro, remove-se o elemento dentro da estrutura, juntamente com o incremento da quantidade de elementos da fila, e após isso retorna no System.out.println() o número -1. Automaticamente, a estrutura dos dados muda dentro da fila, mudando de -1, 25, 5 e 7 para 25, 5 e 7, tornando agora o 25 o valor da primeira posição.

**Figura 3**. Função deQueue.

De maneira geral, nessa classe que implementa a alocação estática existem algumas desvantagens: uma delas é a questão de inserção de elementos ultrapassarem o tamanho máximo do estipulado na criação. Uma solução paliativa seria armazenar os itens em um objeto auxiliar e reinstanciar o objeto atual com um valor máximo maior em tempo de execução, e inserir novamente na estrutura os valores antigos e novos. Porém, torna-se ineficiente, pois se trata de mais processamento para a aplicação e, dependendo do tamanho da nova estrutura, demora demais.

Para aplicações de pequeno porte, que não exijam volumes altos de informação, pode ser trabalhada a fila estática. Uma grande vantagem de trabalhar com o formato estático, tanto Pilha quanto Fila, é a própria aplicação do vetor, que torna o código mais simples de ser implementado e reconhecido.

**COLLECTION CLASS QUEUE – FILA DINÂMICA**

A Fila dinâmica é uma estrutura de dados que implementa os conceitos da fila estática. A diferença se dá conceitualmente no tamanho máximo: como na estática temos o tamanho fixo da estrutura na dinâmica, os elementos podem ser inseridos e removidos sem a estrutura ter um tamanho fixo, tornando o Fila dinâmica um formato melhor para trabalhar em tempo de execução quando não se sabe a quantidade de elementos que deverão ser inseridos.

A interface Queue está dentro da API Collection, implementando a alocação dinâmica com Node. Possui diversas funções já embutidas nela, por exemplo add, remove e peek, funções que poderiam ser implementadas na mão, mas, nesse formato, já estão prontas na interface, otimizando a produtividade ao programar.

Nos imports desse projeto temos, na linha 2 e 3, a primeira referência à Fila dinâmica do projeto, onde se instancia, na linha 8, a interface Queue com o LinkedList e com o tipo Integer, que armazena números inteiros.

Das linhas 11 a 13, adicionam-se os valores 1, 2 e 3 através da própria função add da LinkedList. A linha 15 remove um item da fila, a linha 16 retorna a quantidade de elementos dentro da fila e a linha 17 retorna o valor booleano para a busca do valor 1 dentro da fila com a função contains.

**Figura 4**. Interface Queue.

No uso da Fila são demostradas, no programa, duas formas de navegar nos elementos dela nas linhas 19 a 22 e 24 a 29.

***Clique nas abas para saber mais***

[**PRIMEIRA SOLUÇÃO**](https://sereduc.blackboard.com/courses/1/7.4668.91644/content/_5413555_1/scormcontent/index.html)

Na **primeira solução**, que está nas linhas 19 a 22, utiliza-se a função toArray(), que retorna um vetor da fila e o armazena no vaux[], e no loop faz a varredura pelo vaux. Nessa solução ocorre a transformação de um objeto da API em vetor e pode-se manipulá-lo.

***Clique nas abas para saber mais***

[**SEGUNDA SOLUÇÃO**](https://sereduc.blackboard.com/courses/1/7.4668.91644/content/_5413555_1/scormcontent/index.html)

A **segunda solução**, a mais ideal, é usar o Padrão de Projeto Iterator. Os Design Patterns (Padrões de Projeto) são soluções padronizadas para problemas recorrentes da programação. Em algumas linguagens, deve-se implementar do zero e seguir apenas o conceito; no caso do Java e de algumas outras linguagens há o código pronto, sendo um deles o Iterator.

O Iterator é um padrão de projeto que, através do conceito de orientação a objetos, encapsula a estrutura de dados como uma coleção de elementos e permite percorrer todos da estrutura, ou seja, um padrão de solução para navegação em estruturas que não permitem indexação, como vetor, por exemplo, que ao informar a posição retorna o elemento vetor[posição]. O for-each é uma das variações de Iterator, porém possui algumas funções que precisam ser invocadas para uso; na linha 1 se importa o Iterator.

Na linha 24, o Iterator deve ser instanciado com o nome do objeto que será percorrido com o ".iterator()", onde as classes da API Collection implementam essa solução. Após ser instanciado, cria-se um objeto Iterator da estrutura de dados com o nome filaIterator.

No loop da linha 27, a filaIterator.hasNext() retorna true enquanto houver elementos para navegação na estrutura, e seu acesso é através da função.next(), que retorna o elemento atual do objeto e aponta para o próximo da estrutura, caso houver; e retorna false caso não tenha, encerrando assim o laço de repetição, como se observa na Figura 5:

**Figura 5**. O padrão Iterator.

**Árvores e suas generalizações**

A Árvore é um tipo de estrutura abstrata de dados que comporta-se de forma semelhante a uma árvore, empregando conceitos de raiz e folhas. Também emprega em si fortemente o conceito de Node, ou Nó, como elemento em nível hierárquico. Na hierarquia, comumente os nós podem assumir representações como pai, filhos e irmãos.

O principal Nó da estrutura da árvore é a raiz, o Nó inicial que dá acesso aos demais elementos da estrutura, ou seja, é o ponto inicial em uma navegação, e, a partir dele, pode-se navegar nos filhos e conjuntos de pais e filhos, chamados de conjuntos não vazios ou subárvores, como Node 1, Node 2 e Node n.

**Diagrama 1**. A estrutura Node.

De acordo com critérios de armazenamento, as árvores podem formar ordenação de valores e elementos que façam com que alguns Nós sejam armazenados mais à direita ou à esquerda. No diagrama a seguir, tem-se a árvore Exemplo e inicia-se da Raiz de árvore que chama-se **Node Raiz**.

**Diagrama 2**. Node Raiz.

Os filhos de Node Raiz são Node 1 e Node 2; os filhos de Node 1 são Node 3 e 4, e, obviamente, o Node 1 é pai desses Nodes, e o filho de Node 2 é o Node 5, tornando o Node 2 pai de Node 5. A leitura dessa árvore sempre indica o nome do pai e o nome dos filhos e vice-versa.

Observe que, após os filhos dos filhos, não há mais Nodes; nessa situação, ocorre o conceito de folha, nomenclatura para as extremidades da estrutura da árvore, onde muitos algoritmos utilizam o Node folha para fazer aplicações dentro da árvore.

As árvores podem ser classificadas em hierárquica, diagrama de inclusão, parênteses aninhados e representação encadeada, como se observa nos Diagramas 3, 4, 5, 6 e Figura 6.

* 1
* 2
* 3
* 4
* 5

**Diagrama 3**. Árvore hierárquica.

**Diagrama 4**. Exemplo de inclusão.

**Figura 6**. Parênteses aninhados.

**Diagrama 5**. Adaptação do formato árvore para representação aninhada.

**Diagrama 6**. Adaptação do formato árvore para representação encadeada.

Pode-se observar que a representação do Diagrama 5, passando para o formato da representação encadeada, ficaria como o Diagrama 6.

**// Terminologia da estrutura árvore**

Ás árvores possuem várias propriedades e características que muitos algoritmos utilizam para poder efetuar operações como busca, inserção, remoção, ordenação e classificação. Seguem abaixo suas principais terminologias:

***Clique nos botões para saber mais***

**Nó**

–

Estrutura referenciável que armazena valores dentro de uma árvore, a qual possui apenas um pai, podendo possuir vários “irmãos”; nunca apresenta dois pais ou ligação com seu irmão.

**Arco**

–

Também chamado de vértice ou ligação, é o caminho entre dois Nós pais até os filhos.

**Raiz**

–

É o primeiro elemento Node ou Nó, que dá acesso a todos os elementos da árvore.

**Altura ou profundidade**

–

É o nível do Nó folha ou terminal, somando +1.

**Chave**

–

É denominado chave o elemento que propõe valor ao elemento Nó. Em diversas situações a própria chave do Nó é seu valor.

**Floresta**

–

Conjunto de 0 ou mais árvores.

**Subárvore**

–

São conjuntos de Nós pais após a Raiz, podendo ser consideradas quaisquer Nós contendo sequência de outros Nós.

**Pai**

–

É o Nó que possui filhos ou irmãos; não sendo Raiz, não poderá ser folha; sendo Raiz e não tendo filhos, não poderá ser pai, pois ele mesmo é a folha.

**Filho**

–

É o Nó que possui pai, podendo possuir irmãos ou ser um Nó folha.

**Rótulo**

–

Nome do Nó, geralmente seu dado, valor ou elemento.

**Grau de saída**

–

Número de filhos linkados a ele.

**Grau de uma árvore**

–

É o maior número de filhos de um Nó em toda a árvore.

**Folha**

–

São Nós que não possuem filhos, ou seja, possuem grau de saída 0, também chamados de Node terminal.

**Nó interno**

–

Todos os Nós que possuem filhos ou grau de saída maior ou igual a 1, ou seja, que não são um Nó terminal.

**Caminho na árvore**

–

É a interligação de Nós consecutivos, que traz a relação de hierarquia, por exemplo: “é pai de” ou “é filho de”; é o caminho do Nó Raiz até determinado Nó dentro da árvore.

**Comprimento do caminho**

–

É o número de arco/vértice ou de ligações entre um Nó e outro.

**Nível de um Nó**

–

É a quantidade de arcos da Raiz até o Nó específico; no caso, a Raiz tem 0 arcos, seu nível é 0; sendo um filho da Raiz, possui um arco, então teria nível 1, e assim sucessivamente.

**// Exemplo:**

**Diagrama 7**. **Estrutura Árvore**.

**EXEMPLIFICANDO**

Exemplo de valores para cada propriedade da árvore:

* **Nó**: são todos os elementos Nós da estrutura.
* **Raiz**: A
* **Chave**: do Nó A o valor é A, do Nó B o valor é B, e assim sucessivamente, pois o exemplo não deu uma identificação específica para cada Nó.
* **Grau de saída**: A = 3, B = 1, C = 2, H = 1, E = 0 e I = 0.
* **Grau de uma árvore**: 3, pois A possuí 3 filhos e os demais possuem 2 e 1, que são inferiores.
* **Folhas ou Nó terminal**: E, F, G e I.
* **Nó interno**: A, B, C, D e H, ou seja, Nós que não são folhas.
* **Comprimento do caminho**: A até G é 2 e A até I é 3.
* **Nível de um Nó**: A = 0, B = 1, G = 2 e I = 3.
* **Altura ou profundidade**: Último nível, que é 3 + 1 = 4.

**// Node – Implementação dinâmica**

Nos comandos que implementam a árvore, tem-se a classe Node como classe principal, por ser referenciável e ser moldada ao formato de qualquer topologia, estrutura ou formas de organização.

Essa classe tem como regra armazenar o endereço do Node da esquerda e da direita, formando assim uma ligação entre os filhos para, a partir da ligação com eles, navegar em toda a estrutura.

No código a seguir tem-se o atributo-chave, que armazena a informação ou dado do elemento Node como Object, fazendo com que possa receber diversos valores. Porém, de forma geral, utiliza-se o Integer ao invés de Object na linha 2, como nos exemplos anteriores.

**Figura 7**. Atributo-chave.

A seguir, há uma referência visual da classe Node, em que se observa onde estão sendo feitas as referências à classe da direita, da esquerda e à chave no meio.

**Figura 8**. Classe Node.

**// Vetor – Implementação sequencial**

Nem todas as árvores trabalham com conceito de Node. Em alguns casos, pode-se implementar conceitos de árvore de forma sequencial, em que as chaves das árvores são armazenadas em vetor e os seus filhos representam sempre um múltiplo do seu índice dentro do vetor. A fórmula para acesso aos filhos é utilizada através de seus índices: para acessar os filhos à direita, 2 \* índice, e para os filhos à esquerda, 2 \* índice + 1.

Por exemplo:

**Diagrama 8**. Implementação sequencial.

**ÁRVORES BINÁRIAS**

Uma árvore binária ou binary tree é uma aplicação de estrutura de árvore com regras específicas, como:

* 1

O Nó só pode ter de 0 até 2 filhos;

* 2

Os dois filhos de uma árvore binária são chamados de filho à esquerda e filho à direita;

* 3

As subárvores de uma árvore binária são subárvores da direita ou da esquerda;

* 4

O filho à esquerda de um Nó tem o valor da chave menor que seu pai, e o filho à direita possui o valor da chave maior ou igual a seu pai;

* 5

Uma árvore vazia não contém elementos;

* 6

Todos os elementos da árvore são Nós e arcos.

**Diagrama 9**. Árvore binária.

**// Tipos de árvores binárias**

* **Árvore estritamente binária**: quando todos os Nós da árvore possuem dois filhos ou zero, sendo terminais.

**Diagrama 10**. Exemplo de árvore estritamente binária.

* **Árvore binária completa ou cheia**: são árvores nas quais o último nível é uma subárvore vazia e, por esse motivo, possui a menor altura e todo o Nó terminal no mesmo nível. A altura dessa árvore pode ser calculada pela fórmula.

**Diagrama 11**. Exemplo de árvore binária completa.

* **Árvore binária balanceada**: são estruturas que têm como base a altura das subárvores em 1, e a cada operação na árvore tem-se seu balanceamento, tornando a altura da árvore sempre em 1.

**Diagrama 12**. Exemplo de árvore binária balanceada.

**// Percurso na árvore**

Uma árvore possui diversos tipos de elementos e organizações e, dependendo da forma que se armazenam os valores, pode possuir formatos distintos. No entanto, não há uma indexação para que saibamos os elementos ou estruturas que possui, e apenas através de uma visita ou percurso na estrutura pode-se saber quais elementos e chaves (dados) apresenta.

Existem algumas formas de percorrer os elementos de uma árvore, que são:

* 1

Pré-ordem;

* 2

In-ordem;

* 3

Pós-ordem.

A travessia em **pré-ordem**, ou pré-fixa, inicia-se pela Raiz e percorre os Nós e subárvores à esquerda para depois fazer o mesmo percurso para os filhos da direita, e o respectivo código em Java para percurso.

**Figura 9**. A travessia em pré-ordem.

A travessia em **in-ordem**, ou intra-ordem, percorre a subárvore à esquerda e, ao chegar na folha, volta para a Raiz e em seguida visita a subárvore à direita, assim como o respectivo código em Java para percurso.

**Figura 10**. A travessia em in-ordem.

A travessia em **pós-ordem**, ou pós-fixa, percorre a subárvore à esquerda, depois a subárvore à direita e a Raiz. Por último, o respectivo código em Java para percurso.

**Figura 11**. A travessia em pós-ordem.

**ÁRVORES DE BUSCA**

Uma das maiores aplicações e usos da estrutura Árvore é a busca. Há diversos algoritmos que organizam a estrutura de árvores para maximizar a varredura e melhorar a eficiência do percurso na árvore ao localizar um valor ou chave. A principal árvore para essa função é a árvore binária de busca, a qual, a partir de seu formato de organização, também possui variações como:

***Clique nos botões para saber mais***

**Árvore 2-3**

–

Implementa a árvore binária, porém, na sua inserção, em vez de armazenar apenas uma chave, pode armazenar um intervalo de chaves ou duas chaves, fazendo com que o Node possua dois atributos-chave para armazenar valores de um valor inferior até um superior. Também nos links dos próximos filhos não se tem apenas o direito e esquerdo, mas também o filho do “meio”, tornando a árvore ternária, em vez de binária. A cada inserção, um Nó filho pode possuir mais de um valor na chave e, nesse intervalo, possuir filhos à direita, esquerda e meio, tornando a busca mais veloz por não precisar entrar em novos Nós-Pais para se aprofundar na árvore.

**Árvore AVL**

–

É uma árvore de busca balanceda, ou seja, busca minimizar as comparações entre seus Nós rotacionando ou balanceando a árvore a cada inserção ou remoção, trazendo sempre o melhor formato entre os Nós da estrutura e tornando as alturas das subárvores à direita e à esquerda bem próximas.

**Árvore rubro-negra**

–

Será abordada no tópico Collection Class Tree.

**Árvore splay**

–

É um tipo de árvore binária que transforma os Nós acessados em Raízes através de rotações, fazendo com que elementos mais acessados tornem-se mais rapidamente localizáveis na busca.

**Árvores binárias e de busca**

A Árvore de busca binária (abb) é uma aplicação da estrutura Árvore que armazena suas chaves de forma que, a partir da Raiz, todos os elementos menores que sejam alocados na subárvore esquerda, e todos os elementos maiores sejam alocados na subárvore da direita, fazendo com que as buscas não dependam de algoritmos complexos, mas apenas de comparações entre maior ou menor até o Nó terminal.

**INSERÇÃO**

Considerando a árvore T abaixo, com valores já inseridos, será inserida a nova chave 13. É percorrida a árvore em um formato que faça com que todos os valores menores que o valor a ser inserido estejam à esquerda, como no Diagrama:

**Diagrama 13**. Inserção de valores na árvore de busca binária.

Agora, considere a inseração do valor 8; todos os valores voltados à esquerda serão menores que 8:

**Diagrama 14**. Inserção de valores na árvore de busca binária.

**REMOÇÃO**

A remoção de um Nó consiste em balancear a árvore de forma que os valores à direita sejam maiores que os à esquerda. Considere o Diagrama 15:

**Diagrama 15**. Remoção de Nó na estrutura Árvore de busca binária.

A remoção do elemento 12 faz com que a árvore seja restruturada ou balanceada, no que consiste à busca do menor valor da subárvore à direita, para assumir a posição do valor removido, que no caso é o 13. Observe:

**Diagrama 16**. Remoção de valor binário de busca.

Agora, considere a remoção do 16. Como não possui filhos, a árvore não sofre alterações:

**Diagrama 17**. Remoção sem alterações.

No exemplo da árvore abaixo, a remoção do Nó 6, que não possui filhos à esquerda, faz com que o menor valor da subárvore à direita assuma a posição:

**Diagrama 18**. Remoção e reestruturação.

**BUSCA**

A busca na estrutura Árvore binária faz com que os elementos sejam percorridos de forma que não seja necessário varrer toda a estrutura. Em geral, o primeiro valor de comparação é a raiz, e, a partir dela, busca-se pelo menor ou maior valor na estrutura. Para valores maiores, considera-se a subárvore da direita, e, para menores, considera-se a subárvore da esquerda.

Nesse formato, existe apenas uma comparação entre símbolo de maior ou igual (≥) ou < e, a partir do resultado, percorre-se a subárvore até achar o valor procurado ou retornando nulo, caso percorra até um Nó terminal (que não possui filhos), podendo percorrer nas travessias pré-ordem, in-ordem e pós-ordem.

**COMANDOS DA ÁRVORE DE BUSCA BINÁRIA**

Abaixo, segue a classe Node com as instâncias de esquerdo e direito, e com chave que recebe valores do tipo inteiro iniciando Node de forma null.

**Figura 12**. Classe Node.

Os comandos para a árvore de busca binária no exemplo da classe ArvoreBBin instanciam a classe Node, que possui 99 linhas. Os respectivos conceitos empregados, como a forma de inserir, editar e pesquisar, foram explanados anteriormente no material, mas apresenta-se de forma prática em termos de linguagem de programação Java. A ÁrvoreBBin implementa as operações de inserir, buscar, deletar e percurso in-ordem; a classe que irá manipular essa estrutura é a Prj\_ ArvoreBB.

Nas linhas 9 a 27 encontra-se a função inserirChave, que recebe do programador o valor ou chave que deve ser inserido na estrutura Árvore. Ao ser acionado, chama-se a função private inserirNode com o Nó Raiz atual e o valor a ser inserido na estrutura.

A inserção se dá caso seja o primeiro elemento nas linhas 15 e 17. Caso seja um nível abaixo da Raiz, é observado se o valor é maior ou menor que a chave da Raiz e, caso seja menor é inserido na esquerda; caso seja maior, é inserido na direita em forma de Node.

**Figura 13**. Árvore BBin.

A função de buscar recebe o valor a ser verificado se existe na árvore, é feito uma varredura a partir das verificações dos elementos se maior ou menor que a raiz, e navega-se nas subárvores. É uma função que não utiliza recursão, porém é navegado até o Nó ser terminal enquanto não se localiza o valor. Ao ser localizado na linha 34, faz o retorno de true, caso não na linha 43 após o loop, retorna-se false.

Na linha 46, chama-se a função privada inOrdem() como sobrecarga com parâmetro Raiz, onde se faz o percurso in-ordem da árvore, mostrando primeiramente os valores à esquerda, a Raiz, e em seguida à direita, assim em toda a estrutura da árvore.

**Figura 14**. Função de busca.

A função deletarChave (linha 57) primeiramente recebe o valor a ser deletado e chama a função private deletarNode (linha 62), com o Nó Raiz e o valor a ser deletado.

Nas linhas 71 a 75 é verificado se o valor é menor que o valor da Raiz. Caso seja, é chamada a função deletarNode de forma recursiva, passando por parâmetro a subárvore à esquerda. Caso seja maior, é chamada a função, com a subárvore à direita, ambos farão essa verificação até achar o valor a ser deletado. Caso contrário, não será deletado.

Nessa etapa da função deletarNode das linhas 76 a 95, é verificado se o valor buscado para deletar está dentro da estrutura. Caso esteja, verifica-se os filhos do Nó, nas linhas 79 a 83, e passa-se o Nó filho para tomar o lugar do Nó pai. Nas linhas 85 a 95, faz-se a verificação do menor Nó armazenando no objeto mínimo para tomar o lugar do pai.

**Figura 15**. Função deletar.

Na classe Prj\_ArvoreBB busca-se fazer uso da estrutura montada na classe ArvoreBBin. Nas linhas 7 a 13 é feita a inserção de valores e se chama uma impressão no formato in-ordem na linha 14. Nas linhas 17 a 20, são feitas as exclusões dos Nodes e é chamada novamente a impressão do estado atual da árvore binária. Na linha 23, chama-se a função buscar que retorna boolean true ou false para o valor 8 dentro da árvore binária.

**Figura 16**. Classe Prj\_ArvoreBB.

**COLLECTION CLASS TREE – ÁRVORE**

A classe Tree está dentro da interface Collection e implementa a alocação dinâmica com Node. Possui diversas funções já embutidas nela, como add, remove, contains, entre outras. Essas funções, que poderiam ser implementadas à mão, mas nesse formato já estão prontas na classe Tree, fazem com que se ganhe produtividade ao programar.

No caso, a classe Prj\_ArvoreRN instancia a classe Tree, que implementa como algoritmo a Árvore Rubro-Negro, uma variação de árvore binária. O algoritmo da Árvore Rubro-Negro trabalha com os seguintes critérios:

* O Nó possui um atributo cor que assume os valores preto ou vermelho;
* O Nó Raiz é sempre preto;
* Os elementos possuem chave ou valor;
* Os elementos da árvore são pretos ou vermelhos;
* Os Nós terminais, ou folhas, são pretos e nulos e são ignorados pelo algoritmo;
* Os Nós vermelhos possuem filhos pretos;
* Os Nós pais são de cores diferentes dos filhos;
* Ao inserir o menor valor ou chave, ficará à esquerda enquanto o maior à direita;
* Os percursos nos Nós possuem a mesma quantidade de Nós pretos;
* Implementa fortemente o conceito de balanceamento, onde, a cada operação de inserção ou remoção, a árvore executa rotações para balanceamento e ajuste das cores para manter suas propriedades e equilíbrio.

A linha 6 instancia a TreeSet com o tipo Integer e adiciona os elementos entre as linhas 8 e 13, porém esse algoritmo não aceita valores repetidos, pois são ordenados em tempo de inserção e remoção. Esse algoritmo traz uma baixa performace nas operações de adicionar e remover.

**Figura 17**. Linha 6 com o tipo Integer.

**Grafos**

Os grafos são **estruturas interligadas** que funcionam como um conjunto de elementos e representam uma ligação ou caminho. Funcionam também como uma topologia de rede anel, estrela ou *line*, como um exemplo clássico de entregas em pontos específicos em um determinado território, como cidade, bairro, estado ou país.

Diferentemente das estruturas de dados mais tradicionais, tais como vetor, matriz, lista, fila, pilha e árvores, existem diversas aplicações para grafos. Nesse sentido, eles focam o dado e o dado determina o algoritmo, o comportamento e as formas de armazenamento dessas estruturas. No entanto, quando se trata do grafo, seu foco é voltado ao caminho que se percorre até determinado objeto, e não o objeto em si.

O **grafo** contém dois principais elementos: o **vértice** (que é uma variação de *node*, ou nó) e as **arestas**, também chamadas de **arcos** ou **elos**, que são as interligações de vértices.

**CURIOSIDADE**

As arestas podem conter pontas, fazendo com que sejam direcionadas a um determinado vértice e formando uma “mão única”, ou podem não ter pontas, formando uma “mão dupla”.

**CONCEITOS**

**Adjacência**: ligação de nós por uma aresta ou vizinhos, conforme mostra a Fig. 1. Dessa forma, o nó A é vizinho do nó B.

**Figura 1**. Exemplo de adjacência.

**Caminhos**: sequência de arestas ligadas. Na Fig. 2, por exemplo, é possível ver que o caminho de A até C é ABC e o caminho de A até D e ABD.

**Figura 2**. Exemplo de grafos de caminhos.

**Grafos conectados**: são grafos que possuem caminhos entre seus nós. Por exemplo, na Fig. 3 é possível ver que do F podemos navegar até o A, C e E, pois todos estão conectados.

**Figura 3**. Exemplo de grafos conectados.

**Grafos não conectados**: são grafos que não possuem caminho entre seus nós ou entre todos os seus nós.  Na Fig. 4, por exemplo, é possível perceber que o F não possui caminho até o C ou o E, pois não é vizinho de B, somente de A.

**Figura 4**. Exemplo de grafos não conectados.

**Grafo orientado**: são grafos nos quais a aresta direciona o caminho em um sentido de “mão única”, e os caminhos são diferentes entre suas arestas. O grafo **não orientado**, por sua vez,  é o oposto, ou seja, temos “duas mãos” e o caminho pode ir e voltar pela mesma aresta.

Na Fig. 5, é possível ver um exemplo de **grafo orientado e não orientado**.

**Figura 5**. Exemplos de grafo orientado (A) e não orientado (B).

**Grafo ponderado**: são grafos com arestas com peso, conforme mostra a Fig. 6.

**Figura 6**. Exemplo de grafo ponderado.

**// Código em Java de um vértice ou nó**

O código de um vértice possui propriedades encapsuladas, nome que simboliza seu rótulo ou forma de visualização, índice para demonstrar sua indexação caso seja necessário e o visitado, propriedade de controle que muitos algoritmos utilizam para poder percorrer ou traçar buscas dentro de grafo, visto que ele não possui uma estrutura ordenada ou igual e pode assumir diversas topologias. Seu construtor apenas define seu valor e índice dentro de uma classe grafo.

**Figura 7**. Captura de tela com as propriedades do vértice.

**// Arestas**

Dentro de uma estrutura de grafo, o que interliga os nós ou vértices são as arestas ou arcos. Embora o vértice seja importante, a aresta  é mais importante ainda, pois ela é o caminho ou adjacência entre os nós e, por meio dela, pode-se chegar a, no mínimo, dois nós, e assim por diante.

As técnicas usadas para navegação são: **matriz de adjacência** e **lista de adjacência**.

**// Matriz de adjacência**

O vetor bidimensional, também chamado de matriz, em que o número de linhas representam  as ligações, colunas e quantidades de nós de um grafo, conforme mostrado na Fig. 8.

**Figura 8**. Exemplo de grafo conectado não orientado.

A matriz de adjacência define onde os nós estão interligados a partir de seu nome ou rótulo. No exemplo expressado pela Tabela 1, o nó A está interligado com o B e C. Dessa forma, deve-se colocar na matriz as linhas e colunas de acordo com os nós que estão interligados. O nó A está interligado com B, ou seja, na linha A e coluna B temos 1. Da mesma forma,  o nó A também está interligado com o C, então na linha A e coluna C, temos 1. E linha A com coluna A, por se tratar do mesmo nó, temos o 0, no qual não há uma ligação.

**Tabela 1**. Exemplo de matriz de adjacência.

**// Código em Java da matriz de adjacência**

A classe Grafo\_MA fornece estrutura para navegação e possui algumas funções que trazem resultados que podem servir de base para a tomada de decisões durante o programa.

**Figura 9**. Captura de tela de Grafo\_MA.

A classe inicia com a declaração da matriz de duas posições, denominada matriz[][], e com um vetor de objetos, denominado nos[]. Essas declarações são finais, ou seja, não sofrem mudanças bruscas a partir de sua criação.

A criação se dá na linha 20 com o construtor de Grafo\_MA, que recebe os nós que entram na matriz. Como a matriz possui o mesmo número de linhas e de colunas e esse número é definido pelo total de nós recebidos, então recebe-se o valor de linhas e colunas por meio do nosaux.length.

O **nosaux** é um vetor de vértices recebido por uma classe externa que o cria, popula seus dados e repassa para o Grafo\_MA. Entretanto, ao entrar na classe, os nós são tratados pelo nome "nos", da linha 21.

Após a chamada do construtor de Grafo\_MA, é chamada a função setAresta, linha 25, que recebe por parâmetro a linha e coluna que estão interligadas, setando na variável matriz[linha][coluna]=1, a linha e coluna fornecida pelo usuário ou programa.

Por exemplo, se A está ligada com B, sendo que A é a linha 0 e B é a coluna 1, então na setAresta é: setAresta(0,1), ou seja, a matriz de adjacência tem a ligação da linha A com a coluna B por meio da gravação de 1 no endereço da matriz[0][1]. A função setAresta é a função mais importante da classe, pois ela une os vértices recebidos. Além disso, subentende-se que cada linha e coluna representam um vértice.

A função getCelula, na linha 16, recebe como parâmetro uma linha e uma coluna, retornando o que está gravado nela. A função setAresta, por sua vez,  grava 1 caso possua uma ligação, e o 0 é *default* da própria matriz.

A função getNo, da linha 12, recebe um valor id e retorna ao seu nó correspondente do vetor de vértices. Cada nó recebido dentro do vetor tem um id, que representa sua posição de memória no vetor. Por isso, na hora de criar um vértice, deve-se colocar um id correspondente à posição de memória no vetor.

A função getSize, da linha 8, retorna a quantidade de nós deste grafo.

A função buscarVizinhos, da linha 29, recebe o vértice como objeto, e busca na matriz onde está os seus vizinhos. Para tanto, deve-se navegar nas colunas da matriz, pois a linha que contém o objeto é seu id. Após a navegação entre as colunas, armazena-se em um array os nós vizinhos que contenham ligação com o noaux. A forma de verificar essa ligação é por meio de matriz[][i] !=0, pois sendo diferente de 0 na classe apresentada, pode-se entender que possui ligação. Após fazer a adição de todos os vizinhos do Vértice noaux, ele retorna um ArrayList.

A função imprimirVizinho possui a mesma programação de buscarVizinhos, a única diferença é que ele mostra ao usuário seus vizinhos e não os aloca em uma lista.

**Figura 10**. Captura de tela de Grafo\_MA (continuação).

Após a classe Grafo\_MA com 50 linhas, que trata de um grafo que aloca vértices e arestas e possui funções para navegar e retornar suas propriedades, temos a função Prj\_Grafo, que utiliza a Grafo\_MA como um objeto grafo para manipular em tempo de execução, conforme mostra a Fig. 11.

A classe Prj\_Grafo instancia a classe Vertice, na linha 6, como um vetor de 3 posições: vértice A, B e C com índices 0, 1 e 2, de acordo com suas posições do vértice. Assim, os nós existem e, em seguida, é instanciada a classe Grafo\_MA, passando os nós como parâmetro e criando dentro dele a matriz de adjacência para 3 nós, com 3 linhas e 3 colunas.

Da linha 13 até a linha 18 são feitas as ligações dos nós, em que a linha 0 representa o A, a linha 1 representa B e a linha 2 representa o C. Ainda, na linha 13 é possível estabelecer a ligação da linha A com a coluna B e, na linha 14, a linha A com a coluna C. Na linha 15 é possível ligar a linha B com a coluna A e, na linha 16, a linha B com a coluna C. Por fim, na linha 17 é possível ver a ligação da linha C com a coluna A e, na linha 18, a linha C com a coluna B.

Após as ligações das arestas, na linha 20 ocorre a função imprimirVizinhos, passando o nó A, ou nos[0], como argumento, e trazendo os vizinhos de A.

**Figura 11**. Captura de tela de Prj\_Grafo.

**// Adicionando um nó**

Quando se pensa em uma estrutura em que o mais importante são as arestas e a ligação com os nossos pontos, tratamos de Matriz que possui elementos estáticos. Para adicionar um nó, deve-se crescer a matriz, realocando elementos.  Ao trabalhar com a lista de adjacência, a estrutura de arestas, por trabalhar com o formato de Lista Ligada, não possui este problema.

**// Lista de adjacência**

Dentro de grafos, temos duas formas de representar as ligações entre os vértices: a matriz de adjacência e a lista de adjacência.

Na matriz, há o formato de coluna e linha. Na lista de adjacência, cada nó da lista armazena uma referência a seu vizinho, como uma lista encadeada.

É possível ver a Fig. 12 como exemplo.

**Figura 12**. Exemplo de grafo conectado não orientado.

O exemplo de como o mesmo grafo da matriz de adjacência fica pode ser visto na Tabela 2.

**Tabela 2**. Exemplo de lista de adjacência.

A principal diferença entre matriz de adjacência e lista de adjacência é que cada nó possui seu respectivo vizinho. Em outras palavras, caso tivéssemos 100 nós, teríamos 100 colunas na matriz, tornando a avaliação dos vizinhos mais dificultosa. Em contrapartida, a lista de adjacentes mostra essas informações de forma mais simples. Outro aspecto da lista de adjacência é a questão de alocação do próprio nó na lista, diferente da matriz, que aloca uma numeração 1 e faz uma junção entre colunas, simbolizando o vizinho, a linha do nó e o 1 que une os dois.

**// Código Java para lista de adjacência**

A classe Grafo\_LA, mostrada na Fig. 13, possui como base as declarações do vetor de nós e um vetor de arrays, chamado arestas, nas linhas 5 e 6.

Ao criar a classe Grafo\_LA, passam como parâmetros os nós. Além disso, o vetor de arrays é criado com o tamanho total de nós que foram recebidos.

A função setArestas da classe Grafo\_LA recebe um id, o índice do grafo, e um vetor de Vértices chamados vizinhos. Logo, a aresta na posição informada irá alocar um vetor de vértices.

A função imprimirVizinhos, na linha 17, recebe o vértice a ser procurado,  busca no array as arestas da posição do índice do vértice e imprime em tela.

A função na linha 26 possui a função buscarVizinhos, que retorna os vizinhos do vértice enviado por meio do comando arestas[noaux.getIndice()].  Observe que está se criando um arraylist na linha 28 com o resultado da busca, pois na linha 26 tem como resultado da função o envio de uma arraylist.

**Figura 13**. Captura de tela de classe Grafo\_LA.

A classe Prj\_Grafo\_II, que é uma classe que utiliza a classe Grafo\_LA como objeto para sua estrutura, pode ser vista na Fig. 14.

**Figura 14**. Captura de tela de classe Prj\_Grafo\_II.

Nas linhas 8 a 11, temos a instanciação das classes Vértice em um vetor de 3 posições. Na linha 13, temos a chamada de Grafo\_LA, passando os vetores criados como argumento. Nas linhas 15 a 21, temos a setAresta, recebendo um índice inteiro que simboliza o índice da alocação no vetor "nos". No caso, A é índice 0, B é índice 1 e o C é índice 2.

Após enviar o índice como argumento, temos a passagem da lista de vizinhos, que serão alocados em uma arraylist criada com os nós já instanciados. Para que se tenha uma entrada válida de nós, primeiramente, deve-se ter a criação de um vetor de vértices nos comandos: new Vertice[]{ nos[1], nos[2]}. Esse trecho é um "atalho" no Java para criar um vertor de vértices e pode ser substituído por:

Vertice vaux[]= new Vertice[2]; vaux[0]= nos[1]; vaux[1]=nos[2];      
Ou por Vertice vaux[]=new Vertice[]{nos[1],nos[2]};

Após a criação dos vizinhos como um vetor de vértice, na própria linha 16, apresenta-se um *casting*, transformando o vetor de Vértice em um vetor de Array com o comando Arrays.asList(   ). Após fazer essa conversão de vetor de vértice para array de vértice, cria-se um novo array com estes valores por meio do comando new ArrayList<>.

Por fim, para criar a ligação entre os vizinhos, o comando ficou:

lis.setAresta( 0,   new ArrayList<> (Arrays.asList(  new Vertice[]{nos[1],nos[2]}   ));

Neste mesmo modo, vizinhos para todos os vértices foram criados. Na linha 15, que é o nó A, ficou como vizinho o B (nos[1]) e C (nos[2]). Na linha 17, que é o nó B, ficou como vizinho do A (nos[0]) e C(nos[2]). Na linha 21, que é o nó C, ficou como vizinho do A (nos[0]) e C(nos[2]).

Assim, por meio da setAresta, temos as ligações na lista de adjacência entre os nós, ligando seu id à lista de vizinhos.

Na linha 24, a função buscarVizinhos retorna os vizinhos de C (nos[2]). Na linha 25 temos a impressão dos vizinho de C por meio da função imprimirVizinhos.

**// Buscas**

Em toda a estrutura, a busca é uma das principais operações na estrutura grafo. Sendo assim, temos duas abordagens de busca por profundidade DFS (implementa Pilha) e busca em largura BFS (implementa Fila).

**// Busca por profundidade  - DFS (Depth-First Search)**

Funciona com o seguinte algoritmo: primeiro, no caminho profundidade, navegue para os vizinhos. Dos vizinhos, entre em um ponto de partida, visite o adjacente não visitado, sete-o e coloque-o na pilha. A Fig. 15 e a Tabela 3 mostram a execução de forma mais detalhada.

* 1
* 2

**Figura 15**. Grafo simples e passos para a busca DFS.

**Tabela 3**. Exemplo de busca por profundidade.

**// Código Java**

A classe Buscas possui dois códigos embutidos: DFS e BFS. Ambos usam as funções resetar e getIDVizinhos e também recebem como argumento a matriz de adjacência da classe Grafo\_MA.

A função buscaDFS recebe a matriz de adjacência como argumento no seu construtor da linha 7, fazendo o início da busca. Na linha 8, reseta-se a propriedade visitado do Vértice, pois nos algoritmos de busca é importante saber se um vértice já foi visitado ou não para processá-lo nas buscas caso não tenha sido visitado. Na linha 40, a função resetar recebe o Grafo\_MA como argumento, fazendo esse reset, entrando em cada nó e setando para false a propriedade visitado. Neste caso, pode-se iniciar novas buscas se for necessário.

**Figura 16**. Captura de tela da função de busca DFS.

Na linha 9, inicia-se a pilha que irá armazenar o que precisa ser percorrido e que **não** foi visitado ainda.

O algoritmo inicia com o nó 0 (ou vértice 0) e, na linha 12, seta-se o atributo (ou propriedade) visitado como true, ou seja, a partir deste momento, a busca vai desconsiderar e não colocá-lo na pilha. A pilha insere o nó 0 e, na linha 13, tem-se a apresentação do nome do nó 0 em tela.

Na linha 15, inicia um loop que irá executar enquanto a pilha contiver elementos. Como ela já possui o primeiro elemento, o vértice 0, passa para as linhas de baixo.

Na linha 17, busca-se o id do vizinho do 0, chamando a função getIDVizinhos e enviando como parâmetro a matriz de adjacência, que é o adj, e o índice do nó que está na pilha armazenado.

Na linha 20, ocorre a execução do comando caso o vizinho não seja encontrado. Enquanto no *else*, na linha 22, seta-se a propriedade do vizinho para visitado=true, insere-se o vizinho na pilha e apresenta-se o nome do vértice na tela. Esses comandos são executados até varrerem todos os vizinhos do nó 0, e a busca DFS neste padrão deve ser aplicada em uma situação na qual os grafos não são direcionados e os vizinhos a se buscar devem estar conectados ao nó em questão.

**Figura 17**. Captura de tela da função de busca DFS (continuação).

A função getIDVizinhos, da linha 30, recebe a matriz de adjacência Grafo\_MA adj e o id do nó a ser buscado. Em seguida, verifica se seus vizinhos não foram visitados e se possuem ligação com o id buscado. Para isso, é utilizada a função getCelula, passando o valor do nó atual e a coluna. Caso retorne 1, são vizinhos. Do contrário, retorna 0. Se forem vizinhos, o id do vizinho encontrado é passado para o return j, na linha 34, e o algoritmo aloca o vizinho na pilha do que é solicitado.

**// Busca em largura – BFS (Breadth-First Search)**

A busca em largura consiste em um mecanismo que navega primeiro nos vizinhos para depois implementar a fila nos caminhos. Observe a sequência numérica que se encontra na Figura 18 e na Tabela 4.

* 1
* 2

**Figura 18**.  Exemplo de passos para a busca em largura.

**Tabela 4**. Exemplo BFS.

**// Código Java**

Dentro da classe Buscas sobre o algoritmo DFS, como o BFS, são utilizadas as mesmas funções resetar e getIDVizinho. Dessa forma, tem-se na linha 46 a função buscarBFS, que faz a busca pela largura do grafo.

**Figura 19**. Captura de tela com busca BFS.

Na linha 46, a função recebe como argumento a matriz de adjacência e inicia o reset da propriedade “visitado” dos vértices.

Na linha 49, instancia-se a fila e o primeiro vértice do grafo é enfileirado, alterando seu atributo visitado para true e apresentando-o na tela.

O loop, na linha 49, executa enquanto contiver elementos na fila, ou seja, enquanto o !fila.isEmpty() retornar true.

Na linha 55, pega-se o índice do nó atual e o remove da fila. Na linha 56, pega-se o id dos vizinhos pela função getIDVizinhos, passando a matriz de adjacência e índice atual.

Na linha 58, começa outro loop, que executará enquanto id vizinho for diferente de -1, ou seja, enquanto contiver vizinhos. Seta-se a propriedade do vizinho para true, adiciona o vizinho na fila e o nome do nó é apresentado em tela na linha 62.

A função idvizinhos, na linha 63, chama o novo vizinho de indiceAtual. Entretanto, a função getIDVizinho retorna somente o vizinho que não foi visitado. Nesse sentido, se todos os vizinhos foram visitados, ele sai do segundo loop.

Ao sair do segundo loop, vai para a linha 55, removendo da fila o antigo vizinho e buscando novos na linha 56.

Esse primeiro loop executa enquanto a fila contiver elementos. Todavia, seu critério de parada é a linha 55, momento no qual são removidos elementos da fila. Ao final, quando não tiver elementos, o primeiro loop finaliza.

**// Árvores geradoras mínimas**

A árvore geradora mínima, ou *minimum spanning tree* (*mst*), é um formato em que  um grafo G tem seus elementos percorridos, retornado a um outro grafo com a mesma quantidade de nós ou vértices e com arestas diferentes. Esse mecanismo cria um caminho diferente, semelhante aos galhos de uma árvore, uma vez que suas arestas saem de um “pai” e vão para um “filho”, e assim sucessivamente.

Nesse sentido, aplica-se o algoritmo de busca por profundidade (DFS) ou outros algoritmos para gerar a árvore mínima, embora o DFS seja um dos mais utilizados. No exemplo mostrado na Fig. 20, é possível ver uma “malha” ou um grafo em que todos estão conectados aos nós.

**Figura 20**. Exemplos de árvore geradora mínima.

**Pesquisa de dados (sequencial e binária)**

A pesquisa de dados consiste no desenvolvimento de ferramentas com forma e velocidade para navegar dentro de uma estrutura de dados e buscar um elemento. Nas pesquisas de dados, temos diversos algoritmos, tais como sequencial, binário e hashing.

Assim como a forma de alocação estática e dinâmica são caracterizadas de acordo com seu algoritmo, a pesquisa nas estruturas de dados também são caracterizadas dessa maneira. Entretanto, no que tange à busca, o que define nem sempre é determinado pela regra de negócio ou aplicação do cliente, devendo-se levar em conta o **tempo** para a busca do resultado.

O tempo para busca de um elemento é um dos fatores que influenciam determinada área de estudo da Computação, denominada **análise de complexidade**. Essa área tem como objetivo prever o melhor tempo no qual um algoritmo pode resolver determinado problema sem necessariamente processá-lo.

Dessa forma, o tempo gasto para um algoritmo resolver um problema dentro um cenário muito ruim é chamado de **pior caso**. Um exemplo de pior caso pode ser visto em uma estrutura linear na qual os dados estão totalmente embaralhados e precisam ser ordenados, ou em uma situação onde é necessário buscar elementos dentro de uma estrutura e seu elemento está na última posição de memória. O pior caso sempre é referido como uma fórmula: **O(n) (big O) ou grande O(n)**.

O algoritmo testado é submetido a cálculos por meio de regras da **análise assintótica**, que consiste no conjunto de análises do algoritmo para gerar uma função matemática denominada **N**, representando a quantidade de números a serem passados pelo algoritmo. Por meio dessa fórmula, é possível gerar um gráfico que mostra o comportamento do algoritmo em relação ao tempo e processamento.

A análise assintótica geralmente é composta por funções matemáticas e seu melhor caso é sempre quando se tem uma constante, ou seja, um número em vez de uma variável. Seu pior caso, por outro lado,  é quando se tem uma função quadrática, cúbica ou exponencial.

**Tabela 5**. Funções matemáticas da análise assintótica.

A melhor forma de saber se um algoritmo é eficiente é quando o seu**big O** possui um formato pequeno.

Veja, na Fig. 21, um comparativo entre as principais estruturas de dados. Note que as estruturas que possuem O(1) estão na cor verde e são classificadas como excelentes, enquanto seu pior caso, O(N), é classificado como horrível (em vermelho).

**Figura 21**. Comparativo da Big O das estruturas de dados. **Fonte**: VARUN, 2017. (Adaptado).

**// Busca sequencial ou linear**

Trata-se de uma busca em uma estrutura linear ou vetor que possui uma sequência cuja posição pode variar entre 0, 1, 2... até N.

Além disso, a busca pode ser em formato dinâmico quando se aloca, dentro da estrutura, o endereço do seu próximo até o null, como ocorre nas Listas Ligadas, em que se usa a estrutura Nó ou Node.

**Figura 22**. Captura de tela de busca sequencial.

**// Busca binária**

Trata-se de uma busca particionada em uma estrutura ordenada na qual um elemento pesquisado começa no meio da estrutura e, a partir da metade, ocorre uma análise para decidir se a pesquisa seguirá antes ou após o meio da estrutura.

**Figura 23**. Captura de tela de busca binária.

**Cálculo de endereço – espalhamento (hash)**

O **hash** é uma técnica de programação que visa garantir uma locação dinâmica para um dado.  Em outras palavras, por meio de algoritmos hash, é possível receber um dado e garantir se ele terá apenas um número dentro de uma alocação vetorial.

No exemplo demonstrado na Fig. 24, há um objeto 1, 20 e 66 que, ao passar em um algoritmo hash, “converte” esses objetos em uma numeração única. Trata-se de uma posição de um vetor ordenado.

**ASSISTA**

O algoritmo de hash é muito utilizado em assinaturas digitais, para garantir segurança e confidencialidade nas mensagens enviadas por empresas e pessoas de maneira geral. Para saber mais sobre o assunto, assista ao vídeo*Assinatura Digital e Hash*, disponível no Youtube.

[**ASSISTA**](https://www.youtube.com/watch?v=UlRCVihN3pE)

**Figura 24**. Exemplo processo hash.

Além disso, o inverso também é verdadeiro:  por meio do algoritmo hash é possível fazer com que o número da posição se torne o valor original.

No Java, o hashcode de um objeto é uma função numérica que traz um int em relação a um objeto. Esse int é um hashing em relação ao JVM do próprio Java. Na Fig. 25, é possível ver vários objetos instanciados da linha 6 a 10 todos com o valor 1.

**Figura 25**. Captura de tela de Código Java para hash em objetos instanciados.

No entanto, como armazenam de forma diferente, o código hash pode ser visto na Fig. 26, visto que trata-se do mesmo número, mas para objetos Integer, Float, Double, Long e String.

Ao instanciar o HashObjetos na linha 11, é possível ver que o código hash 1829164700 é mostrado na saída da linha 18. Nesse **formato**, observa-se que cada objeto tem um hash.

**CURIOSIDADE**

O Java instancia os objetos encontrados entre as linhas 6 e 10 de forma direta.

**Figura 26**. Captura de tela de saída do código HashObjetos.

Ainda, a forma tradicional de instanciação é:

Integer inteiro = new Integer(1);  
Float flutuante = new Float(1.0f);  
Double doublevar = new Double(1.0d);  
Long longvar = new Long(1);   
String string = new String("1");

Todavia,  as versões mais atuais do Java fazem a instanciação no seguinte formato:

Integer inteiro = 1;  
Float flutuante = 1.0f;  
Double doublevar = 1.0d;  
Long longvar = new Long(1);  
String string = "1";

De todo modo, a instanciação realizada produz o mesmo resultado, pois é normal que as versões da Linguagem Java facilitem alguns códigos para os programadores, assim como as implementações de API’s, como o API Collection ou o Iterator para navegar em objetos.

**HASHING**

A técnica hash aplica o algoritmo hashing, que funciona como um algoritmo criptográfico e transforma o dado em um índice por meio de operações aritméticas. Este índice, denominado chave, é obtido por meio da fórmula:

**hashcode = Valor % Total de Posições**

O valor de hashcode é um inteiro que recebe o cálculo do valor passado e total de posições de uma alocação dinâmica ou estática.

A função em Java fica:

public static int getHashCode(int valor, int tamanho){  
        int hashcode = valor % tamanho;  
        return hashcode;  
    }

Trata-se da operação aritmética chamada de **operação modular**. A operação modular é uma operação que envolve números naturais e que reinicia a partir de um valor determinado. Por exemplo, um relógio de ponteiro dá 12 voltas, ou seja, possui 12 horas. Quando o ponteiro chega no número 12, a contagem reinicia.

No hashing, a operação modular traz um range do tamanho do vetor ou total de posições. Ao ultrapassar o tamanho total do vetor, a contagem começa novamente.

Caso seja no formato de alocação estática, as posições do vetor já existem, e se forem alocados em formato dinâmico, as posições irão ser inseridas conforme tempo de execução, estabelecendo um limite para os valores que podem armazenar.

Para o exemplo de um cálculo hash em tempo de execução, a Fig. 27 mostra um código que possui uma função que recebe um valor, a quantidade de um vetor e retorna a seu hash correspondente.

**Figura 27**. Captura de tela do código em Java da técnica hash.

Na linha 5 da Fig. 27, a função getHashCode recebe como argumento o valor a ser armazenado e o tamanho do suposto vetor a ser armazenado. Na linha 6, é feita a operação modular entre o valor e a posição do vetor que deverá ser alocada. Na linha 12, há um loop que inicia de 0 a 11, chamado função getHashCode, passando pelos números de 0 a 10, com o valor 7 simulando um vetor com 7 posições. A função getHashCode fornece a posição de alocação para o valor dentro de um vetor de 7 posições, conforme pode ser visto na Fig. 28.

**Figura 28**. Captura de tela de saída do código HashCode.

Observe, na Fig. 28,  que o “Valor:” varia de 0 até 10, e suas posições também variam entre 0 e 6, representando, no caso, um vetor de 7 posições (contando o 0). Se o valores fossem apenas de 0 até 6, por uma coincidência os valores também são armazenados de forma  sequencial e crescente, mas a partir do valor 7 temos já uma **colisão**. O valor 7 fica na posição 0 e assim sucessivamente, até 10 na posição 3.

**// Colisão**

A **colisão** é um problema derivado do hash que, ao passar pelo algoritmo de hashing, os diferentes valores podem assumir as mesmas posições. Logo, o valor é convertido em uma célula. A Tabela 6 apresenta a situação de efeito de colisão.

**Tabela 6**. Efeito de colisão.

Para resolver este tipo de problema, utiliza-se uma técnica chamada separate chaning, ou **encadeamento separado**, na qual é implementada no vetor de hash (ou tabela hash) uma lista encadeada ou *linkedlist*. Assim, a partir de uma colisão, cria-se uma lista ao lado da posição para armazenar os valores daquela posição, conforme mostra Tabela 7.

**Tabela 7**. Tabela hash com implementação de lista ligada.

Esse formato de código implementa a alocação estática, pois necessita de um vetor de tamanho fixo. Para alocação dinâmica, utiliza-se a collection HashSet.

Ao utilizar o **encadeamento separado**, dependendo da quantidade de colisões pode-se ter mais valores em uma lista na posição do que necessariamente na tabela toda.

**Tabela 8**. Exemplo de encadeamento separado.

O ***load factor*** (tamanho ou fator de carga) é uma propriedade de tabela hash que fornece parâmetros para saber o tamanho limite da tabela. Caso o número de colisões seja maior do que o esperado, a solução é criar uma tabela maior que armazenará os valores da tabela antiga para a nova. Essa técnica é chamada **rehash**.

O ***linear probing***(exploração linear) é a capacidade de armazenar os valores na posição sequencial caso haja uma colisão. Por exemplo: um hash de um número é 123. Caso haja colisão nesta posição, o valor da hash para o novo item será 124.

**HASH PARA STRINGS**

O computador não diferencia letras de números da mesma forma que os seres humanos. O que o faz processar letras diferentes de números são os códigos que estão por trás dos números.

Em baixo nível, todas as letras usam os caracteres da tabela ASCII (**American Standard Code for Information Interchange**), que consiste em uma tabela na memória do computador que faz o processamento binário de cada letra, número ou caractere especial. Entretanto, para maior flexibilidade, converte-se do binário para decimal ou hexadecimal em tempo de execução.

Por exemplo: a letra A vale 65 em decimal e 41 em hexadecimal. Além disso, há diferenças entre maiúsculas e minúsculas. É possível ver, na Tabela 9, os códigos para todas as letras.

**Tabela 9**. Tabela ASCII (American Standard Code for Information Interchange). **Fonte**: SUGAI, 2015. (Adaptado).

Como o hash trabalha somente com números, as letras são convertidas nas tipagens ou objetos strings. Além disso, cada valor de letra é convertido para números ASCII.

Para o valor string de nome=”Ana” , o ASCII de Ana é:

Entretanto, para números que se somam, o ASCII pode fazer a colisão facilmente, pois a somatória nos fornece um número pequeno e, para evitar a colisão, usa-se a potência que torna a somatória grande e diminui drasticamente a colisão nas strings e char. Isso é possível com a seguinte fórmula:

**char[0] \* 27n-1+ char[1] \* 27n-2 + ... + char[n-1]**

Sendo que:

* o número 27 é considerado o alfabeto de 26 letras somado a um caractere de espaço (igual a 27 elementos);
* N é o length da string, ou quantidade de caracteres da palavra;
* o char[posição] é o char da cadeia string, que será multiplicado pelo 27 elevado a N-1.

Para aplicar em Java, é preciso transformar o objeto string em char com a função **.charAt(posição)**. Por exemplo:

   String nome="Ana";  
        long hash= (long)(nome.charAt(0)\* ( Math.pow(27, nome.length()-1)));  
        hash += (long)(nome.charAt(1)\* ( Math.pow(27, nome.length()-2)));  
        hash += (long)(nome.charAt(2)\* ( Math.pow(27, nome.length()-3)));  
        System.out.println("O hash de “+nome+” é : " + hash);

O código hash da string Ana é 50452, fazendo com que palavras derivadas entre A até An estejam em uma faixa de quase 50 mil posições, que podem ser alocadas.

Nesse código, usa-se o long, pois os números podem ser grandes e o casting para conversão das operações de exponenciação (Math.pow(x,y) é a função de exponenciação do Java que eleva x a y) que saem em double para long por meio dos símbolos (long). O charAt(posição) retorna o char, mas a sua conversão para int é automático (nesta operação) e em nível binário. Além disso, é efetuado o cálculo se houver necessidade de conversão para int.

No entanto, o hash traz números muito grandes e, para uma tabela que geralmente possui poucas posições, usa-se o hashing para poder aplicar o valor da chave Ana, que é 50452 em um tamanho de vetor pequeno.

Por exemplo:

Considerando a função getHashCode com o valor 50452 e com o tamanho 7 de vetor, ele seria armazenado na posição 3.

public static int getHashCode(int valor, int tamanho){  
        int hashcode = valor % tamanho;  
        return hashcode;  
    }  
System.out.println(HashCodeExemplo.getHashCode(50452, 7));

**HASHSET NA COLLECTION**

O algoritmo hash desenvolvido no formato estático utiliza o vetor e absorve suas propriedades, que são o valor fixo de posições, uma tipagem e um valor para a posição. Para a alocação dinâmica da tabela hash, utiliza-se o conceito de Node e de Lista Ligada.

A classe **HashSet** implementa a interface collection, conforme mostra a Fig. 29; na linha 3, importa-se o pacote útil que contém a classe HashSet, a interface Set. Diferentemente da interface List, que implementa o formato sequencial, a Set é uma interface que implementa conjuntos hash.

**Figura 29**. Captura de tela de Classe HashSet.

Quando se usa a interface Collection, as funções são padronizadas independentemente da estrutura utilizada. Elas são:

* Add – insere elementos;
* Remove – remove elementos;
* Contains - busca elemento na estrutura.

No código da linha 9 ocorre a instanciação. Nas linhas 11 e 15 são inseridos 5 valores. Além disso, é feita a impressão da estrutura hash das linhas 20 a 23, com sua impressão via for-each na função imprimir.

**// Collection HashMap**

Na collection HashMap, diferentemente da HashSet, devem ser inseridos a chave e o dado. Para isso, deve-se usar o comando put em vez de add. O add, em geral, recebe apenas o dado a ser inserido, o put, a chave e o valor, fazendo um mapa de dados com o índice fornecido.

**Figura 30**. Captura de tela de collection HashMap.

Neste formato de código, pode-se buscar e inserir pelo índice em vez de somente pelo dado, dando mais autonomia para o programador em relação à forma como indexar seus dados.

Para a navegação dos Maps, é preciso primeiro buscar um conjunto dos índices que retorna um SET por meio da função keySet() na linha 22 e, a partir deste conjunto de índices, buscar o dado de forma individual, pelo método get(), na linha 25, passando o índice como argumento.

Para uma navegação, torna-se mais lento, pois temos este processo. Entretanto, para as buscas, torna-se mais veloz, pois fornece o índice e ele fornece os dados como na linha 19, com a função containsKey().

Caso seja preciso localizar um dado por si em uma Map, pode-se usar a função containsValue(), com o valor a ser localizado, retornando true ou false para a localização do valor.