**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS**

GABRIEL VARGAS BENTO DE SOUZA
NILSON DEON CORDEIRO FILHO

# TRABALHO PRATICO 01 – AEDS III

Relatório de Implementação do Algoritmo

Belo Horizonte - MG
2023

## 01. Musica.java

Para o trabalho, foi utilizada a base de dados de Músicas do Spotify do seguinte link: https://www.kaggle.com/datasets/saurabhshahane/spotgen-music-dataset. Ela possuia mais de um arquivo csv e muitos registros, então, para adaptar ao trabalho, a base foi modificada unindo aluns arquivos e apagando alguns atributos. Assim, foi possível montar um arquivo mais uniforme e coerente com a proposta do trabalho, mantendo, claro, os atributos pré-requisitados no enunciado.

A classe Musica possui os atributos presentes na base de dados, bem como o int id e o boolean lapide. Foi escolhido representar essa lapide como boolean por ser um tipo de dado escrito com um único byte, economizando, assim, espaço no registro.

Além disso, tem os métodos de leitura e de atribuição para ler e salvar os registros como um objeto do tipo Musica. Foi interessante criar os getters e setters para os atributos, pois, com a dinâmica de separação de pastas e de packages, o trabalho ficou mais fluído e organizado.

Por fim, tem-se os métodos fromByteArray() e toByeArray() para converter os atributos em um fluxo de bytes e serem salvos em arquivo binário. Destaca-se que existem toByteArray() e toByteArray(int tamanho): o primeiro é o conversor padrão e, nele, obtém-se o tamanho de forma calculada; o segundo, por sua vez, tem a finalidade de ser utilizado no update quando deve ser mantido o tamanho original caso o novo registro seja menor ou igual ao que havia anteriormente.

Ademais, vale considerar que estas funções escrevem, dentre outras coisas, os atributos que são String em arquivo com a função writeUTF(), porém, antes, é gravado um inteiro para corresponder ao tamanho desta String. Para a String de tamanho fixo, foi implementado como array de char[], assim, é utilizada a função writeChar().

## 02. IO.java

Durante a dinâmica de leitura de dados do teclado no trabalho, foi percebido a possibilidade de o usuário digitar tipos de dados não coerentes com o solicitado. Por exemplo, sendo esperado um inteiro, o usuário digitar uma String que não pode ser convertida para tal. Assim, o programa estava apresentando erros. Então, foi decidido a implementação da “IO.java”, que serve mais como entrada do que como saída de dados, justamente para tratar os erros de tipo e não prejudicar a execução.

Essa classe apresenta apenas as funções de ler inteiro e ler linha (String), porém, a partir dela, é possível converter para os outros tipos de dados utilizados.

## 03. CRUD.java

Para a efetiva primeira parte do TP01, foi implementada a classe CRUD que serve, justamente, para a principal manipulação do usuário com a base de dados. É importante frisar que foi tomado o cuidado para que todos os métodos apresentados abaixo possuíssem tratamento de exceções, de tal maneira que o programa não será encerrado com erro de digitação do usuário.

Assim, o primeiro método é o carregarCSV(), que, como o próprio nome induz, tem por finalidade abrir o arquivo csv e popular a base de dados que foi chamada de “Registros.db”. Tal como recomendado, foi gravado um cabeçalho, no início do arquivo,

com um inteiro correspondente ao último id criado para controle e para servir como base no método de create.

Levando em conta a possibilidade de o banco de dados “Registros.db” já ter sido criado, caso o programa já encontre o arquivo, é solicitado que se decida entre manter o já existente e criar um novo. Como é de se supor que, na maioria dos casos, não é de se esperar que apague o registro com as modificações feitas anteriormente, a opção 1 faz com que o registro se apague, todavia a 2, tal como qualquer outro número, letra ou símbolo que possa ser digitado cancela a operação. Foi optado não exigir que o usuário digite apenas a opção 2 neste caso, diferentemente dos outros menus mais à frente.

A segunda funcionalidade do CRUD é o create(). Embora simples, este método lê o último id salvo no cabeçalho, “chama” uma função do “Musica.java” para leitura dos atributos pelo teclado e salva a nova música com o novo id no final do banco de dados. Em seguida, altera o cabeçalho do arquivo para o novo id. Tal como mencionado anteriormente, erros de digitação de tipo por parte do usuário não param o programa. Assim, os inteiros lidos incorretamente são tidos como 0 (zero), a data é tida como 01-01-0001 e a String, como “”.

A próxima funcionalidade implementada é read (int idProcurado). Por meio dela, é possível fazer uma busca sequencial pelos registros, procurando a partir do id e parando ou ao chegar no fim de arquivo ou ao chegar na música procurada. É trazido para a memória principal uma música por vez que é lida como fluxo de bytes, convertida no objeto música e comparada; caso encontrada, é printada na tela. Para chamar esse método, tem-se o read() que lê o id e faz a busca.

Em seguida, há o delete (int idProdurado) que é praticamente igual ao método anterior, com a única diferença de que, quando encontrar o id, ao invés de mostrar na tela, marca o registro como falso, settando a lápide para true e reescrevendo no mesmo lugar do arquivo.

A funcionalidade seguinte é o update (int idProdurado) que mantém a mesma estrutura também. Contudo, ao encontrar o id que se deseja, é aberto um menu para o usuário selecionar qual atributo deseja alterar, com exceção do id e da lápide, foi decidido possibilitar alterar todos para, caso digite alguma informação incorreta, não ter de deletar e recadastrar a música.

Nesse sentido, uma observação importante é a de que, ao se alterar um registro, o novo pode se enquadrar em dois casos: I) o tamanho do novo se torna maior que o atual, sendo assim, este é redirecionado ao final do arquivo e o lugar atual é marcado como lápide; II) o novo tamanho é menor ou igual, sendo assim o registro em fluxo de byte é recriado, mas, dessa vez, passando o tamanho anterior como parâmetro. Isso é feito utilizando uma função citada anteriormente “toByteArray(int tamanho)”. Ademais, ressalta-se o cuidado para quando o registro já havia sido alterado anteriormente para um tamanho menor, porém, ao se alterar de novo, embora o tamanho possa aumentar, caso o novo fique ainda sim menor do que era o original, o registro permanece no mesmo lugar.

Como requisitado, o crud já estava todo montado, porém foi decidido acrescentar duas novas funcionalidades para incrementar o trabalho: abrirMusica() e salvarTXT().

Por se tratar de uma base de dados de músicas do spotify é de grande valor a possibilidade de o usuário poder ouvir essa música. Para tanto, o método criado é semelhante ao read(), mas, ao invés de mostrar na tela, é salvo o uri da música e, por meio de uma função para abrir link na web, a música é aberta no navegador padrão.

Para linux e mac, a classe “ProcessBuilder” foi a escolhida, pois não demonstrou erros de execução nos testes. E, para o Windows, “Runtime”, visando evitar possíveis problemas.

A outra função incrementada ao sistema é a de salvar todos os registros como String em arquivo txt. Vale ressaltar que segue os mesmos passos dos outros métodos, como o de read(), contudo todas as músicas são escritas em arquivo.

**04. MinHeap.java**

Essa classe é responsável pela construção de um heap mínimo para servir como base para a distribuição da Intercalação Balanceada com Seleção por Substituição. O heap é construído de maneira padrão, a não ser por um detalhe exclusivo da ordenação em questão: ter dois arrays, sendo um para o heap propriamente dito e um para a ordem de prioridade de cada elemento.

Todos os métodos de inserção, remoção e heapify dependem do responsável por obter a prioridade de um registro. Esta é obtida de acordo com o elemento presente no topo do heap: se maior, a prioridade se mantém a mesma, pois ainda pode ser ordenado no mesmo arquivo, mas, sendo menor, a prioridade é acrescida em uma unidade, assim só será considerado para o próximo arquivo temporário.

Vale, ainda, frissar o int atributo, presente na classe, serve para controlar qual será o atributo de ordenação: id, nome ou data de lançamento. Estes foram escolhidos a dedo para serem cada um de um tipo de dado, bem como extremamente significativos para o tipo de registro que está sendo trabalhado.

**05. QuickSort.java**

Esta classe é relativamente simples também por se tratar de um algoritmo de ordenação em memória primária bem difundido. Esse algoritmo possui complexidade O (lg n) no melhor e no caso médio. Embora o pior caso seja O ($n^2$), não é uma possibilidade muito frequente, pois o arquivo tende a estar praticamente ordenado, o que faz com que tenda ao melhor caso.

Tal como no minHeap, exite o int atributo para decidir qual será o atributo utilizado na ordenação.

**06. ComumSort.java**

A primeira parte deste algoritmo de ordenação, consiste no construtor padrão da classe. Nele, é parametrizável o número de caminhos (arquivos) e o número de registros ordenados em cada caminho. Foi optado que, como a base possui 22725 registros, o NUM_REGISTROS fosse igual a 1500 e o NUM_CAMINHOS, igual a 4 no construtor padrão. É possível outros valores, inclusive foram testados e são escolhidos pelo usuário, mas limita-se que tenham, no mínimo, 2 arquivos distintos e que o seguimento ordenado seja maior que zero, obviamente.

Em seguida, tem-se o método boolean distribuicao (int atributo). Ele serve para distribuir os NUM_REGISTROS pelos arquivos desejados. Inicialmente, são criados os arquivos temporários para se gravar os registros, porém, no início do “arqTemp0.db”, é

salvo o id correspondente ao último criado, tal como feito na funcao carregarCSV() e na create().

Depois, enquanto houver arquivo e dentro de um loop variando de 0 até o NUM_CAMINHOS, os registros são trazidos para a memória principal e ordenados com o quicksort de acordo com a quantidade parametrizada. Assim, são escritos nos arquivos temporários intercalando entre eles.

Vale lembrar que o int atributo serve, também, para decidir qual o atributo utilizado no quicksort para a ordenação. E que a função é boolean, pois devolve true apenas se a distribuição ocorrer corretamente, evitando que o programa trave.

O outro método extremamente importante é o int intercalacao (int atributo, int numIntercalacao, boolean paridade).

- O atributo indica se será ordenado pelo id, pelo nome ou pela data;
- O numIntercalacao indica em qual intercalação está e, por conseguinte, quantos resgistros em sequência estão ordenados;
- A paridade serve para indicar qual arquivo temporário está mais ordenado e qual deve ser descartado para escrever nele novamente;
- A função retorna a quantidade de arquivos criados.

A primeira parte corresponde à criação dos arquivos temporários e, no primeiro, salvar o ultimo id pelo mesmo motivo citado anteriormente (sempre que acabar a intercalação, o registro que estará ordenado será o primeiro a ser criado).

Depois, enquanto não chegar o fim de todos os arquivos, eles são visitados por ponteiros, e a primeira música é lida e salva em um array para carregamento inicial. A menor música, de acordo com o atributo, é salva no novo arquivo. O processo se repete novamente, porém agora o único registro que será lido é aquele que teve sua última música gravada no outro arquivo. A condição de parada de leitura no arquivo é ter atingido a parte ordenada (calculada a partir do NUM_REGISTROS e do numIntercalacoes) ou o fim de arquivo. Quando isso ocorre, o arquivo é simplesmente pulado até que todos tenham sido lidos a quantidade indicada. Esse loop todo se repete até que tudo tenha sido trazido para os novos arquivos. Existe um contador no método que serve para contabilizar a quantidade de arquivo gerados entre [1 e NUM_CAMINHOS].

Estes dois processos (distribuição e intercalação) são executados no método ordenar. Nele, os registros são distribuídos uma única vez e, depois, intercalados, variando a paridade e o numIntercacoes.

Por fim, quando apenas um arquivo temporário for criado, o “Registro.db” é excluído, e o temporário (que já possui o ultimo id no cabeçalho) é renomeado. Além disso, os temporários extras são todos deletados.

**07. TamanhoVariavelSort.java**

Esta ordenação é uma variação da anterior, seguindo todos os passos iguais, exeto por uma única verificação. Enquanto o comumSort tem duas possibilidades de parada na leitura do arquivo (contador de parte ordenada e fim de arquivo), o TamanhoVariavelSort não utiliza o contador; ao invés disso, o arquivo temporário é lido até que a música lida não esteja ordenada em relação à anterior. Caso isso aconteça, a leitura é ignorada, o arquivo é marcado como inválido por um instante e o ponteiro volta

um registro (como se não tivesse lido). Todo esse processo se repete até que o fim de arquivo chegue.

**08. SelecaoPorSubstituicaoSort.java**

A terceira ordenação é semelhante à anterior na parte de intercalação com tamanhos variados. Entretanto, a fase de distribuição em memória principal se difere por não utilizar o quickSort, mas sim o minHeap.

Sobre o construtor, ele é semelhante por também ser parametrizável, porém o NUM_REGISTROS também serve como a capacidade do heap, mesmo que não seja uma potência de 2. Assim, o construtor padrão possui, também, 1500 registros.

A distribuição no heap ocorre trazendo para a memória primária um registro por vez até que o heap fique completo. Em seguida, remove o topo do heap (que é o menor valor), salva-o em arquivo e insere um novo. Como dito anteriormente, o heap tem o int prioridade, este atributo serve para direcionar a qual arquivo temporário que deverá ser gravado o registro, respeitando, assim, o NUM_CAMINHOS. Para tanto, é feito (prioridade % NUM_CAMINHOS) para garantir, com o módulo, o direcionamento correto.

**09. OrdenacaoExterna.java**

Esta classe é, praticamente, um menu para os três tipos de ordenação e para a escolha do usuário sobre qual será a desejada e utilizando qual atributo para se ordenar. Nela, lê-se do teclado a escolha e, assim, a devida função com seus parâmetros é chamada. Vale frisar que é solicitado que o usuário digite o número de caminhos e o número de registros; todavia, caso seja inválidos, o padrão será 1500 divididos em 4 arquivos, não solicitando um nova leitura.

**10. Main.java**

A classe principal do programa foi trabalhada para deixar a mais limpa possível. Nela, encontra-se um menu principal para o contato entre o usuário e o programa. Por meio dela, é possível executar todos os métodos e manipular, de todas as formas explicitadas, o banco de dados.

**11.MakeFile**

Para uma maior organização, o trabalho foi dividido em pastas e em packages, sendo assim, está extremamente mais fácil o entendimento sobre cada parte do programa. Nesse sentido, visando facilitar ao máximo a execução, foi criado um Makefile para compilar todas as classes na devida ordem e redirecionando para uma pasta bin os arquivos “*.class”. Para compilar, basta utilizar o comando “make run”.