Pontifícia Universidade Católica de Minas Gerais
ICEI - Curso de Ciência da Computação
Disciplina: Algoritmos e Estruturas de Dados III
Prof. Zenilton Patrocínio Jr.

Trabalho Prático de Algoritmos e Estrutura de Dados III - Entrega 2

Model - Magazine Abakós - ICEI - PUC Minas

Henrique Castro e Silva[1]
Leonardo Caetano Gomide[2]

Resumo

Este trabalho tem como intenção desenvolver um sistema que faça manipulação de arquivos em memória primária e secundária. O sistema em questão, consiste em um cadastro de prontuários para um empresas de planos de saúde, com opções de inserção, alteração, exclusão e impressão. Conforme estipulado pelo cronograma da disciplina, este trabalho será composto por três entregas complementares, cada entrega responsável por entregar diferentes artefatos. Este documento é referente à primeira entrega, que é composta por diferentes artefatos que foram desenvolvidos pelos alunos. Esses artefatos incluem a criação de um arquivo mestre, o desenvolvimento de um sistema que faça todas as operações de CRUD (criar, ler, atualizar e deletar) e produção de um documento de texto. Para o desenvolvimento dos diferentes requisitos técnicos foram desenvolvidas diferentes classes e funções. A formatação do arquivo mestre, a finalidade e as especificações de cada classe e o funcionamento do sistema em geral serão abordadas neste documento.

Palavras-chave: Algoritmos e Estrutura de Dados. CRUD. Manipulação de arquivos.

[1]Programa de Graduação em Ciência da computação da PUC Minas, Brasil– henrique.silva.1281914@pucminas.br
[2]Programa de Graduação em Ciência da computação da PUC Minas, Brasil– lcgomide@sga.pucminas.br

Abstract

This assignment focused on developing a system that may manipulate files in primary and secondary memory. The system consists of a registration of medical records, including insertion, updating, exclusion and printing options. According to the discipline's stipulated schedule, this assignment will be composed of three complementary deliveries, each responsible to deliver different features. This document refers to the first delivery, which is composed by different features developed by the students. These features include the creation of a master file, a system's development, which include all CRUD (create, read, update, delete) operations, and generation of textual document. In order to develop all the technical issues, different classes and functions were developed. The master file's formatting, the purpose and specifications' of each class and the overall systems' running will be discussed in this document.

Keywords: Algorithms and Data Structure. CRUD. File manipulation.

1 INTRODUÇÃO

O trabalho aqui apresentado consiste em um sistema para gerenciamento de prontuários. Para o desenvolvimento desse sistema, foi escolhida a linguagem de programação Java, que permitiu a criação do CRUD completo, a interação com os arquivos de dados e implementação da solução. Nas sessões subsequentes, será abordado a estruturação do projeto e a funcionalidade de seus artefatos.

2 ESTRUTURA DE ARQUIVOS

No intuito de tornar o acesso aos arquivos do projeto mais claro e simples, os diferentes arquivos foram segmentados em diferentes diretórios seguindo a estrutura abaixo.

```
root
├─ /dados
└─ /src
   ├─ /dao
   ├─ /main
   ├─ /manager
   └─ /model
```

2.1 Dados

No diretório "dados"encontram-se os arquivos de dados. Quando o usuário do sistema cria um novo arquivo mestre de dados, ele ficará disponível nesse diretório.

2.2 Source

No diretório "src"encontram-se todos os arquivos de código fonte desenvolvidos e necessários para o funcionamento do sistema. Esse diretório por sua vez possível diferente ramificações, conforme será exposto a seguir.

2.2.1 Dao

O diretório "dao", abreviado do inglês *Data Acess Object*, encontram-se as classes responsáveis por fazer o acesso aos dados por meio das classes *Manager*. Nessas classes encontram-se os métodos de CRUD. Contudo as classes nesse diretório possuem um nível de abstração superior a outras classes, assim sendo, quando executado algum método, elas são responsáveis de interpretar o objeto passado e delegar as respectivas funções necessárias para outras classes.

2.2.2 Main

Nesse diretório encontra-se a classe *App*, que é a primeira classe a ser executada quando o projeto é executado. Por meio dela que ocorre a interação do usuário com o sistema.

2.2.3 Manager

As classes encontradas no diretório "manager"são as classes que gerenciam os arquivos de dados dos sistemas, tentando ficar o mais próximo do baixo nível. Ou seja, elas que fazem operações com Bytes, sendo essas operações a escrita, a sobrescrita e a leitura de vetores de Byte.

Nelas também já encontram-se métodos responsáveis por manusear os metadados do arquivo, guardar informações sobre os tamanhos do registro e o caminho para o arquivo mestre.

2.2.4 Model

Por fim, o diretório "model"é responsável por armazenar as classes que definem os objetos do sistema.

3 IMPLEMENTAÇÃO

Uma vez exposta a estruturação do projeto, pode-se detalhar a implementação dos métodos implantados.

3.1 Incialização

Antes de ter acesso ao sistema, o usuário deverá primeiro informar se ele já utilizará dados existentes ou se ele criará um novo arquivo de dados. No primeiro caso, é informado somente o caminho para a pasta onde estão os dados, e os atributos são obtidos lendo os dados contidos nos cabeçalhos desses arquivos. Caso opte pela segunda opção, diferentes atributos são requisitados, incluindo:

- *documentFolder* - A pasta onde serão guardados os arquivos de dados do sistema.
- *dirProfundidade* - Profundidade incial do arquivo de diretório.
- *registersInBucket* - Número de registros que serão guardados nos *Buckets* do índice.
- *dataRegisterSize* - Tamanho em Bytes de um registro.
- *dataNextCode* - Código inicial para registros (opcional).

Seguida da obtenção desses metadados, é iniciada a etapa de criação dos arquivos de Diretório, Índice e Mestre, nomeados respectivamente *dir.db*, *idx.db* e *data.db*.Nota-se que quando um novo conjunto de arquivos é criado, o índice é populado com *n* Buckets em branco, onde *n* correponde à profundidade do diretório elevada ao quadrado.

3.2 Modelo

As classes de modelo são as estruturas basicas do projeto, definindo os objetos do sistema. Sendo elas:

3.2.1 Prontuario

Em nossa implementação da entidade prontuário optamos pelos seguintes atributos:

Tabela 1 – Atributos

Nome	Tipo de Dado	Tamanho (Bytes)
codigo	int	4
nome	String	variável
dataNascimento	java.util.Date	8
sexo	char	2
anotacoes	String	variável

Quando o Prontuário é codificado para ser inserido no arquivo de dados, os atributos são salvos nessa ordem utilizando os métodos da classe *java.io.DataOutputStream*, no caso da *dataNascimento*, é obtido a quantidade de milissegundos desde 1970 e codificado como um

long, no momento dessa codificação se o registro codificado estourar o tamanho pré-definido de registro do arquivo, uma *IndexOutOfBoundsException* é estourada.

3.2.2 Bucket

Em nossa implementação da entidade Bucket optamos pelos seguintes atributos:

Tabela 2 – Atributos

Nome	Tipo de Dado	Tamanho (Bytes)
profundidade	int	4
length	int	4
emptyLength	int	4
data	HashMap	8*length

Para o atributo *profundidade*, o valor é definido baseado no campo *bucketSize* definido pelo usuário. Já o campo *emptyLength* corresponde a quantos registros ainda cabem dentro do Bucket. No momento da inserção de um novo valor no Bucket, o valor desse atributo seja 0, ocorrerá uma exceção do tipo *IndexOutOfBoundsException*.

Quando salvo em formato de Bytes, são escritos os primeiros 3 atributos e para cada par, chave e valor, no atributo *data* são escritos os dois valores como int. Caso o atributo *emptyLength* seja maior que 0, são populados *n* pares do valor *-1*, onde *n* é a diferença entre o campo e 0.

3.2.3 Directory

Em nossa implementação da entidade Diretório optamos pelos seguintes atributos:

Tabela 3 – Atributos

Nome	Tipo de Dado	Tamanho (Bytes)
profundidade	int	4
buckets	int[]	variável

Dentro dessa classe que encontra-se a função Hash e a função de estender o arquivo. Apesar dos dados do Diretório estarem salvos em um arquivo, optamos por deixa-lo em memória e apenas alterar o arquivo em memória secundária quando ocorrerem mudanças.

3.3 Arquivos de Dados e Managers

Conforme abordado na seção 3.1, uma vez inicializado o sistema existem 3 arquivos salvos em memória secundária, cada qual possui uma classe manager específica que é responsável pela leitura dos arquivos e processamento de seus dados.

Todos os arquivos seguem estruturas semelhantes, onde os primeiros Bytes sao correspondentes a metadados, administrados por seus respectivos Managers, e o restante sendo o conteúdo dos arquivos.

Vale ressaltar também a existência da classe *ManagerManager*, que nada mais é a classe "erenciadora dos gerenciadores", ou seja, ela faz a comunicação entre o diretório, o índice e o arquivo mestre. Por meio desta que é feito o CRUD, presente na seção seguinte.

3.4 CRUD

A classe *App* fica responsável de "interagir"com o usuário e servir de menu de opções. Uma vez que o usuário passe valores de input válido, um objeto da classe *ManagerManager*, nomeado *manager*, e *ProntuarioDAO*, nomeado *dao*, ficam encarregados de finalizar a operação.

3.4.1 Create

Para a inserção de um registro, *dao* chama a *manager* passando o código e o objeto já codificado em bytes, com os quais, é realizada a inserção, que utiliza o atributo *firstEmpty* do *dataManager* para reaproveitar o espaço, caso esse atributo não seja *-1*, o *firstEmpty* é atualizado com o próximo valor da pilha, contido no endereço a ser reutilizado, e o objeto é inserido nessa posição que é retornada à *manager*. Com esse valor, é calculado em qual bucket esse registro será inserido encontrando a sua posição no diretório por meio da função Hash, utilizando o campo código como chave.

Nessa operação podem haver algumas exceções, como o Bucket apontado pelo Hash ainda não ter sido inicializado ou não haver espaço nele. Em ambos casos, manager resolve os conflitos e atualiza os arquivos necessários.

3.4.2 Read

Para leitura, a chave é passada para o objeto *manager* que fica responsável de encontrar o Bucket onde ela foi armazenada e em sequência encontrar em qual posição o registro foi salvo no arquivo mestre.

Uma vez com essa posição, o arquivo mestre é acessado e o vetor de bytes é passado ao objeto *dao*, que por sua vez converte esses bytes em um objeto.

3.4.3 Update

O mecanismo de atualização faz uso do índice para encontrar rapidamente o prontuário especificado e sua posição no arquivo. Caso encontrado, é utilizado o *DbManager* para sobrescrever o prontuário antigo com o objeto recebido como um vetor de bytes pela *dao*.

3.4.4 Delete

O método de deleção faz uso de outro método para sua operação, a atualização, que recebe o objeto em um vetor de bytes a ser alterado, mas quando chamado pelo *deleteObject*, é passado somente a lápide preenchida e um inteiro para compor a pilha de deleção. O inteiro é obtido pela váriável *firstEmpty* do *dataManager*, que é substituída pela posição atual do objeto no arquivo de dados.

3.5 Índice

Neste trabalho fazemos uso de um índice de Hash extensível para melhorar o desempenho para a busca de um objeto dentro do arquivo mestre. Para tal, sempre que um objeto é inserido, atualizamos também um arquivo com a posição de cada um dos objetos do sistema e seus respectivos códigos, organizados de acordo com o resultado de uma função Hash, que será descrita na seção 3.5.1.

3.5.1 Diretório

O diretório é um arquivo usado para redirecionar um objeto para um bucket, por meio de uma função Hash combinada a uma tabela com os resultados dessa função e a posição do bucket, consistente para qualquer objeto e preferencialmente uniformemente dividida no espaço de busca. Para a nossa aplicação, simplesmente utilizamos o código do Prontuário e obtemos seu módulo de 2^n, sendo n a profundidade atual do diretório, garantindo que sempre que a função seja executada para uma mesma chave, o bucket correto sempre será encontrado com uma complexidade *O(1)*.

3.5.2 Buckets

Os Buckets são as estruturas que armazenam as referencias para os objetos no arquivo de dados, essas estruturas armazenam uma chave e a posição do arquivo, tal que sempre que

uma chave é recebida pelo bucket correto, por meio do Diretório, é possível encontrar o objeto procurado.

Esta estrutura possui uma limitação da quantidade de chaves que pode armazenar, sendo que quando uma chave tenta ser inserida após o limite ser excedido, uma expansão precisa ser executada, que pode ser de dois tipos: de bucket, e de diretório.

Para o primeiro caso, a profundidade do bucket é menor que a do diretório, fazendo com que existam no mínimo duas posições do diretório apontando para um mesmo bucket. Então fazemos com que cada segunda posição aponte para um mesmo novo bucket, e todos os objetos do bucket original precisam ser reorganizados entre esses dois novos buckets, possívelmente ativando uma nova divisão, caso todos os itens se mantenham em somente um dos dois buckets.

Para o segundo caso, o existe somente uma instância do bucket dentro do diretório, forçando sua expansão, que é simplesmente aumentar sua profundidade duplicar o diretório antigo tal que para toda posição $p > 2^{n-1}$ o bucket que será apontado será o apontado pela posição p - 2^{n-1}. Depois que essa operação é executada, basta fazer uma divisão de bucket.

4 BENCHMARKING

Nesta seção abordamos os resultados dos testes realizados pelo grupo. Para realizar os teste, utilizamos o sistema operacional Windows em um computador que não estava completamente dedicado, ou seja, estava executando outras tarefas em segundo plano, no entanto, para evitar variações por conta de uso, os testes foram realizados enquanto o computador não era utilizado. Utilizamos um HDD de 1TB e um SSD 128GB de capacidade e um processador i5-7200U 2.5GHz. No momento dos testes o HDD tinha cerca de 200 GB não utilizados enquanto o SSD possuía cerca de 5GB livres. Para minimizar o impacto do pouco espaço de armazenamento livre, após cada teste, todos os dados relacionados a ele são deletados.

Tabela 4 – Testes Realizados

Qtd. testes	Qtd. Registros	Prof. Inicial Dir.	Registros/Bucket
5	20.000 (~10 MB)	2	62
5	20.000 (~10 MB)	8	62
2	200.000 (~100 MB)	2	62
2	200.000 (~100 MB)	2	126
2	200.000 (~100 MB)	2	510
2	200.000 (~100 MB)	8	62
2	200.000 (~100 MB)	8	126
2	200.000 (~100 MB)	8	510
1	500.000 (~250 MB)	8	62
1	500.000 (~250 MB)	8	510
1	5.000.000 (~2 GB)	8	62
1	5.000.000 (~2 GB)	8	510

Realizamos 26 testes diferentes, todos tanto em um HDD quanto em um SSD, variando diferentes parâmetros, os parâmetro eram o número de prontuários inseridos, variando entre

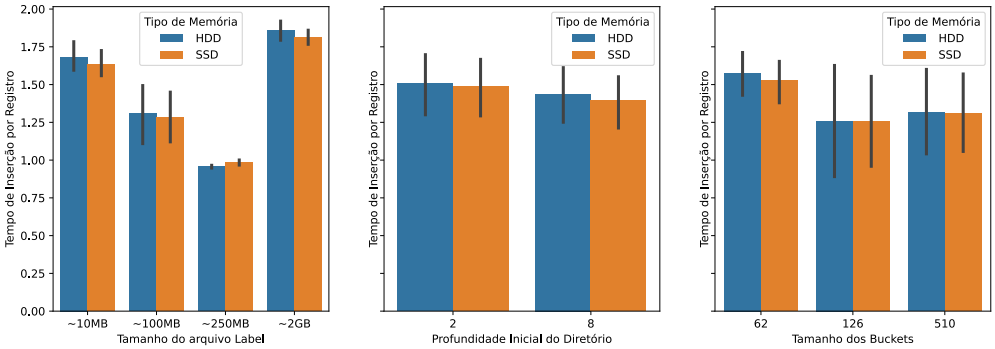

Figura 1 – Comparação de Inserção

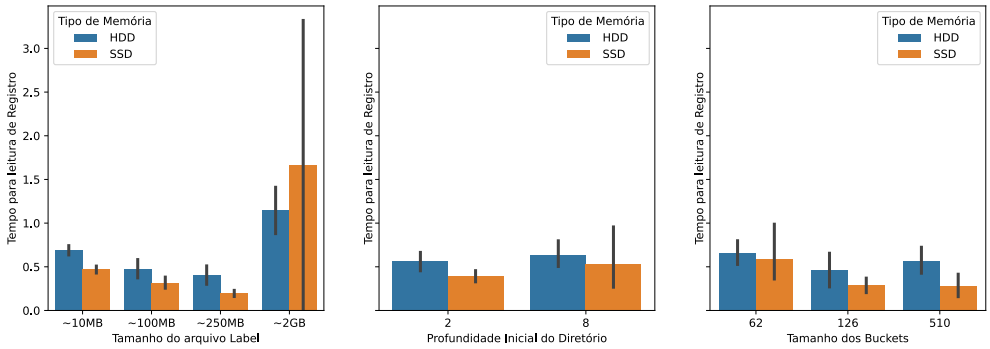

Figura 2 – Comparação de Leitura

20.000 (10MB), 200.000 (100MB), 500.000 (250MB) e 2.000.000 (2GB); Profundidade inicial do diretório, 2 e 8; O número de registros por bucket (tamanho do bucket), 62, 126 e 510 registros, esses números foram escolhidos para que os buckets possuam tamanhos próximos de 512 bytes, 1MB e 4MB. Assim sendo, testamos algumas das combinações possíveis dos parâmetros (tomar como referência Tabela 4), e para cada teste fizemos o benchmarking de operações de leitura, escrita, atualização e deleção, sendo feitas 100 de cada tipo de operação e tirando a média de todas no final.

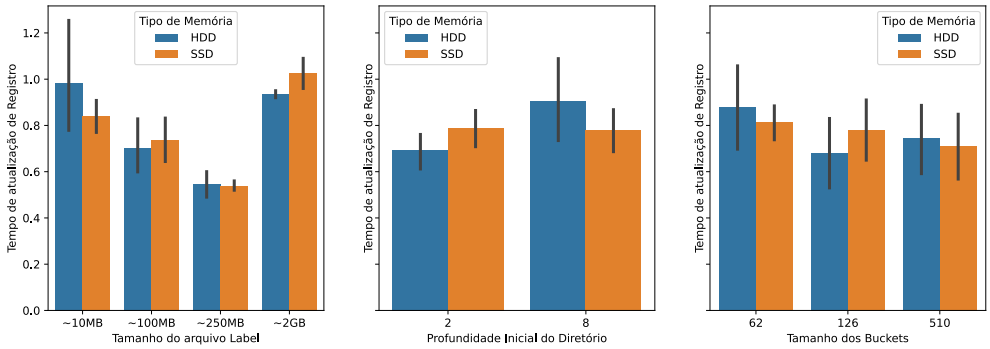

Figura 3 – Comparação de Atualização

Feitos os testes, nos é retornado um CSV com os dados destes testes, com o qual reali-

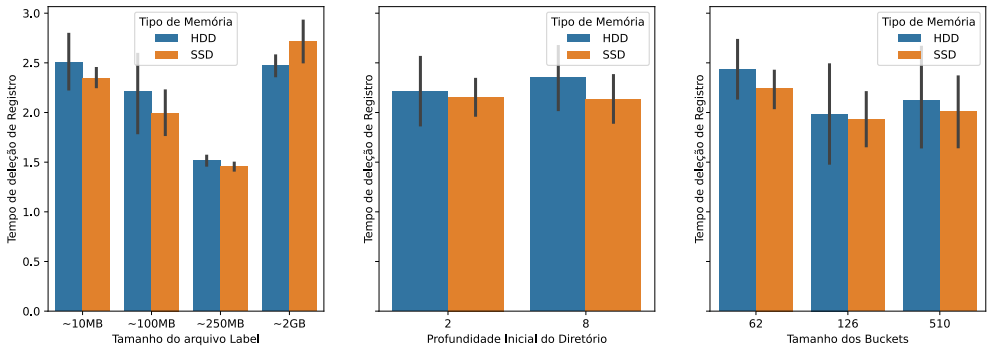

Figura 4 – Comparação de Deleção

zamos uma análise com Pandas e Seaborn na qual foram gerados as figuras 1, 2, 3 e 4.

Notou-se que para todos tipos de operações, os tempos médios gradativamente diminuem entre os tamanhos de 10MB, 100MB e 250MB, contudo no arquivo de 2GB todas operações apresentaram maior tempo médio, tanto em SSD, quanto em HDD. Outra observação geral possível é que o tamanho do bucket quando sendo 62 registros apresentou o maior tempo médio em todas operações, contudo a mudança de 126 para 510 não apresentou ganho significativo.

Fora as obervações gerais, foi possível observar um comportamento estranho em algumas situações, como o tempo de inserção no arquivo de 2GB, que em média levou mais tempo no SSD do que em HDD, fora uma grande variância entre os dados; outro ponto seria que em muitos testes notou-se uma diferença pequena entre o desempenho no HDD e no SSD. Supomos que isso aconteceu devido ao espaço disponível no SSD, contudo seriam necessários mais testes para validar esta hipótese.