PUC Minas
A B A K Ó S
Instituto de Ciências Exatas e Informática

Revista Abakós do Instituto de Ciências Exatas e de Informática Pontifícia Universidade Católica de Minas Gerais*

Model - Magazine Abakós - ICEI - PUC Minas

João Paulo Maia de Paula[1]

Resumo

Este projeto tem como objetivo realizar a implementação de um sistema de cadastro de prontuários, utilizando hash extensivo, para uma empresa de planos de saúde fictícia. Este sistema foi implementado em linguagem Java e possui um menu que permite acesso aos diferentes métodos que fornecem opções de inserção, alteração, exclusão e impressão. Os dados são armazenados em memória secundária em arquivos com extensão *"db"*, de forma a serem persistentes e poderem ser acessados entre as instâncias de uso sem perda de informação ou necessidade de uma nova entrada de dados a cada nova utilização. A criação dos arquivos de armazenamento é feita de forma opcional, possibilitando a criação de novos ou a utilização dos arquivos já existentes. Ao longo de todo o código foram realizados comentários, de forma a facilitar a leitura e interpretação da lógica utilizada para a solução dos problemas enfrentados e solucionados. O código foi desenvolvido pelo próprio autor deste projeto com auxílio de fontes diversas.

Palavras-chave: Java. LaTeX. Prontuário. Programa. Hashing.

*Artigo apresentado ao professor Zenilton Kleber Gonçalves do Patrocínio Júnior

[1]Curso de Ciência da Computação, Brasil– joao.paula.1278444@sga.pucminas.br

Abstract

This project's objective is to implement a medical record system, utilizing extendible hashing, for a fictional health insurance company. This system was implemented in Java and utilizes a menu that allows access to the different methods that provide the options for insertion, editing, exclusion and printing. The data is stored in a secondary memory on files with the extension .db, in such a way that they are persistent and may be accessed between use instances without loss of information or requiring new data input from the user every time the program is used. The data files creation is optional, making it possible to generate a new files or utilizing the previously existing ones. Across the whole code there are comments, intended to facilitate reading and interpretation of the logic used to solve problems faced and solved. The code has been developed by the author itself with aid from different sources.

Keywords: Java. LaTeX. Medical. Record. Software. Hashing

1 INTRODUÇÃO

O projeto tem como objetivo a criação de um sistema de cadastro de prontuários médicos com informações persistentes salvas em memória secundária. Conta com métodos de inserção, alteração e exclusão dos dados os quais são salvos em arquivos com extensão *db* sendo estes lidos entre as instâncias de uso e as informações relevantes são carregadas em memória principal e atualizadas conforme necessidade. A utilização da linguagem Java se mostrou uma excelente escolha devido a ela possuir diversos métodos pré-existentes de manipulação, serialização e desserialização de dados.

2 DESENVOLVIMENTO

Foi desenvolvido um código em Java para a inserção, edição, remoção e impressão de prontuários médicos contendo como parâmetros: marcador de lápide, CPF, nome, data de nascimento, sexo e área para anotações, sendo esta última um parâmetro inserido pelo usuário na criação do arquivo. A metodologia de desenvolvimento intencional foi semelhante a de *SCRUM*, criando metas a serem entregues em cada momento de desenvolvimento. Com o objetivo de introduzir o aprendizado deste método juntamente ao projeto. Também tendo em vista a limitação de ser um único autor do programa.

2.1 Prontuário

O programa possui uma classe denominada *"pront.java"* que tem como objetivo ser o objeto dos prontuários médicos. Quando instanciada ela é utilizada para escrita e leitura, recebendo a entrada de dados pelo console para gravação no arquivo e também a informação desserializada para os dados serem armazenados em memória principal, atualizados e manipulados conforme necessário.

2.1.1 Variáveis

O prontuário conta com variáveis *boolean* (lapide), *int* (CPF) e *String* (nome, dataNasc, sexo e anotacoes) para as informações pertinentes, dois construtores sendo um vazio para inicialização das variáveis e outro que recebe um parâmetro para cada variável permitindo que o objeto seja inicializado com valores específicos.

2.1.2 Métodos

Como métodos possui: *"public String toString"* para impressão de um prontuário específico o qual retorna as variáveis já formatadas para serem impressas diretamente no console.

"public byte[] getByteArray" utiliza as classes *"ByteArrayOutputStream"* e *"DataOutputStream"* para a serialização e escrita das informações em arquivo, retorna o vetor de *bytes* para poder ser utilizado,

"public void setByteArray" utiliza as classes *"ByteArrayInputStream"* e *"DataInputStream"* para desserialização e leitura dos dados vindos do arquivo.

"public void Inserir" para escrita em uma posição específica, recebendo parâmetros de posição e tamanho para este fim. Trabalha em conjunto com o método *"getByteArray"*.

"public int Ler" análogo ao método "Inserir", porém, para leitura, este método retorna diferentes valores para poderem ser usados como forma de debug e manipulação de dados fora da classe. Trabalha em conjunto com o método *"setByteArray"*.

2.2 Índice

O programa possui duas classes denominadas *"Bucket.java"* e *"SubBucket.java"* que juntas tem como objetivo ser o objeto do índice. Quando instanciadas, elas juntas são utilizadas para escrita e leitura, recebendo a entrada de dados referente a quantidade de entradas (SubBuckets) através do console. Também são utilizadas para gravação no arquivo e a informação desserializada de forma que os dados serão armazenados em memória principal, atualizados e manipulados conforme necessário.

2.2.1 Variáveis

O índice conta com variáveis *int* (profLocal, qtdPorBucket e qtdSBUsados) referentes ao seu cabeçalho através da classe *"Bucket.java"*, possuindo também diversas variáveis de controle e manipulação de forma a facilitar as informações como tamanho do cabeçalho e de suas entradas. Enquanto a classe *"SubBucket.java"* conta com variáveis *boolean* (lapide) e *int* (CPF e endereco) como seus atributos.

2.2.2 Métodos

Como métodos em comum ambas as classes possuem: *"public String toString"* para impressão de seus atributos, o qual retorna as variáveis já formatadas para serem impressas diretamente no console.

"public byte[] getByteArray" utiliza as classes *"ByteArrayOutputStream"* e *"DataOutputStream"* para a serialização e escrita das informações em arquivo, retorna o vetor de *bytes* para poder ser utilizado,

"public void setByteArray" utiliza as classes *"ByteArrayInputStream"* e *"DataInputStream"* para desserialização e leitura dos dados vindos do arquivo.

"public void Inserir" e *"public void InserirEndereco"* para escrita em uma posição específica, recebendo parâmetros de posição, endereço e tamanho para este fim. Trabalha em conjunto com o método *"getByteArray"*.

"public int Ler" análogo ao método "Inserir", porém, para leitura, este método retorna diferentes valores para poderem ser usados como forma de debug e manipulação de dados fora da classe. Trabalha em conjunto com o método *"setByteArray"*.

Para a classe *"SubBucket.java"* existe: *"public int LerEndereco"* quase idêntica a *"public int Ler"* salvo que recebe o parâmetro end ao invés de tam de forma a ser feita a leitura em um endereço direto no lugar de um cálculo.

Para a classe *"Bucket.java"* existem: *"public static int buscaQtdSB"* que busca a quantidade de entradas por *bucket* (valor passado como parâmetro pelo usuário durante a criação dos arquivos) e retorna este valor para poder ser utilizado.

"public static SubBucket[] iniciaVetorSB" para inicializar um vetor de *"SubBuckets"* nos construtores de forma que o tamanho do vetor é utilizando a quantidade de *"SubBuckets"* passado pelo usuário.

2.3 Diretório

O programa possui duas classes denominadas *"Diretorio.java"* e *"SubDiretorio.java"* que juntas tem como objetivo ser o objeto do diretório, de forma muito parecida as classes que se referem ao índice. Quando instanciadas, elas juntas são utilizadas para escrita e leitura, recebendo a entrada de dados referente a profundidade global através do console, com objetivo de ser utilizada junto ao hash. Também são utilizadas para gravação no arquivo e a informação desserializada de forma que os dados serão armazenados em memória principal, atualizados e manipulados conforme necessário.

2.3.1 Variáveis

O diretório conta com uma variável *int* (profGlobal) referente ao seu cabeçalho através da classe *"Diretorio.java"*, possuindo também diversas variáveis de controle e manipulação de forma a facilitar as informações como tamanho do cabeçalho e de suas entradas. Enquanto a classe *"SubDiretorio.java"* conta com variáveis *int* (valor e endereco) como seus atributos além de uma (tamCabecalho) para controle.

2.3.2 Métodos

Como métodos em comum ambas as classes possuem: *"public String toString"* para impressão de seus atributos, o qual retorna as variáveis já formatadas para serem impressas diretamente no console.

"public byte[] getByteArray" utiliza as classes *"ByteArrayOutputStream"* e *"DataOutputStream"* para a serialização e escrita das informações em arquivo, retorna o vetor de *bytes* para poder ser utilizado,

"public void setByteArray" utiliza as classes *"ByteArrayInputStream"* e *"DataInputStream"* para desserialização e leitura dos dados vindos do arquivo.

"public void Inserir" e *"public void InserirEndereco"* para escrita em uma posição específica, recebendo parâmetros de posição, endereço e tamanho para este fim. Trabalha em conjunto com o método *"getByteArray"*.

"public int Ler" análogo ao método "Inserir", porém, para leitura, este método retorna diferentes valores para poderem ser usados como forma de debug e manipulação de dados fora da classe. Trabalha em conjunto com o método *"setByteArray"*.

Para a classe *"Diretorio.java"* existem: *"public static int calculaQtdPaginas"* para calcular a quantidade de páginas do diretório, sendo este 2 elevado a profundidade global.

"public static SubDiretorio[] iniciaVetorSD" para inicializar um vetor de *"SubDiretorios"* nos construtores, de forma que o tamanho do vetor definido pela quantidade de páginas já calculado anteriormente.

2.4 Menu

O menu possui opções listadas de 1 a 7 em *loop* com testes para a seleção inválida como números não listados ou caracteres que não sejam numerais. Em cada opção existe uma chamada para a classe e método correspondente, como por exemplo no caso 1 "Criar arquivo"chamando "CArq.criaArquivo()"ou 2 "Inserir registro"com "CRUD.InserirReg()". Com exceção do caso 7 "Sair"que encerra o programa.

2.4.1 Variáveis

Possui variáveis de entrada, que utiliza a classe *"Scanner"* para receber valores digitados pelo usuário para a seleção das opções do menu e *boolean* para manipulação do *loop*. Também conta com variáveis que recebem os valores contidos no cabeçalho do arquivo para fins de teste.

2.5 Criação dos arquivos

A criação dos arquivos de armazenamento é feita pela classe *"CArq.java"* que checa se existem arquivos anteriores e caso sim deleta estes antes de criar novos, garantindo assim a integridade dos arquivos. É usado um *loop* para garantir que a entrada do usuário referente a profundidade global, quantidade de entradas por *bucket* e tamanho da área de anotações seja um valor numérico. Ao final são criados três arquivos, "diretorio.db", "indice.db"e "prontuarios.db"na pasta "dados"com a escrita de seus respectivos cabeçalhos, sendo que ambos o diretório e índice já possuem suas páginas e entradas criadas com valores baseados nas entradas do usuário. Também é criado um 4º arquivo, este sem cabeçalho ou entrada de dados, de nome "logTempo.db"que é utilizado para armazenar os tempos de execução para inserção, busca e impressão.

2.5.1 Variáveis

Possui variável de entrada, que utiliza a classe *"Scanner"* para receber valores digitados pelo usuário referentes a profundidade global, quantidade de entradas por *bucket* e o tamanho do campo de anotações. E variáveis de controle *boolean* e *int* para manipulação do *loop* e armazenamento das entradas do usuário. Os arquivos são criados utilizando uma variável de *"RandomAccessFile"*.

2.6 Manipulação de dados

A manipulação de dados é feita através de classe *"CRUD.java"*, sendo esta a maior classe do programa pois possui os métodos que manipulam *Strings* e controlam a inserção, edição, deleção e impressão dos dados.

2.6.1 Variáveis

Possui diversas variáveis como de entrada, que utilizam a classe *"Scanner"* para receber valores digitados pelo usuário, para seleção de opções e entrada de dados, variáveis de controle e manipulação com os valores dos tamanhos dos campos do registro, páginas e entradas e variáveis do cabeçalho para a atualização dos dados após manipulação e alteração destes. Também conta com uma variável chamada de "qtdAutoInsere"que controla quantos *loops* serão auto inseridos.

2.6.2 Métodos

Como métodos possui *"public void profGlobalInicial()"* que é utilizado para inicializar e atualizar a profundidade global e quantidade de páginas para serem utilizados por outros métodos.

"public static int vHash" que retorna o valor hash a partir de parâmetros *int* e *string* utilizando *hashCode*, utilizando do resto de divisão pela profundidade global, assegurando que o valor será utilizado corretamente pelos *buckets*.

"public static void InserirReg" para inserção de dados nos arquivos, primeiro lendo o cabeçalho e armazenando seus valores nas respectivas variáveis e em seguida solicita as informações do prontuário ao usuário. Com as informações do registro em memória efetua um teste para verificar se existem registros com lápides que tenham valor *false* (deletados), caso positivo insere o registro no primeiro valor vazio atualizando também a quantidade de valores vazios no cabeçalho, caso contrário insere o registro no final do arquivo. Durante a inserção é armazenado o endereço do prontuário de forma a ser registrado no *bucket* correto. Por fim quando inserido o registro é feito uma atualização no cabeçalho da quantidade de CPFs. Em seguida é manipulado o diretório, de forma que a função hash é usada para buscar qual *bucket* deverá ser direcionado este novo prontuário, armazenando o endereço dele para ser buscado diretamente. Por fim é feita a atualização da lápide, CPF e endereço do primeiro *SubBucket* vazio daquele *Bucket* sendo atualizado no cabeçalho do *Bucket* a quantidade de *SubBuckets* ocupados.

"public static void AutoInserirReg" para inserção automática de dados nos arquivos. Tem a funcionalidade idêntica ao método "InserirReg", pois é uma cópia deste com o acréscimo de um *loop* que utiliza a variável "qtdAutoInsere"para definir quantos registros serão inseridos.

"public static void EditarReg" para a edição e atualização dos campos de anotações dos prontuários. Faz uma busca utilizando o CPF inserido e recebe a entrada de dados do usuário, atualizando aquele prontuário.

"public static int RemoverReg" para a remoção de prontuários, sendo utilizada uma remoção lógica na qual é feita a alteração da lápide para *"false"* de forma que o programa passa a considerar aquele registro como vazio. É feito uma busca utilizando o CPF inserido e caso o prontuário existe é feito um pedido de confirmação de exclusão.

"public static void ImprimirArq" para a impressão de todos os arquivos em console. É feita a leitura do cabeçalho do arquivo, sua impressão e em seguida o resto de suas informações, *SubDiretorios* para o diretório, *SubBuckets* para o índice e por fim os prontuários. Sendo utilizado uma combinação de *"System.out.println"* e método *"toString"* de cada classe de forma a ser feita a formatação dos dados em console.

3 TESTES E SIMULAÇÕES

3.1 Hardware e SO

O computador no qual todos os testes foram executados possui as seguintes especificações técnicas:

- Processador: Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
- RAM: 16,0 GB DDR4 3000 MHz
- SO: Windows 10 64-bit

3.2 Parâmetros e metodologia

Os testes inicialmente foram executados em um SSD com 1000 e 10000 registros com valores de p = 8 (profundidade global), n = 1000 (número de entradas por *bucket*) e m = 1000 (tamanho das anotações). Porém, a partir de 10000 registros os testes se mostraram inviáveis devido ao tempo de inserção muito extenso.

Os testes foram executados utilizando a opção 6 do menu, que apaga os arquivos antigos e cria novos após receber parâmetros para p, n e m (k é definido como 1000 para todos por padrão). Em seguida, preenche o arquivo mestre com prontuários de teste utilizando um *loop*, associando o endereço de cada prontuário a seu respectivo *bucket* e *subbucket*. Por fim, busca e imprime os registros, índice e diretório, armazenando os tempos de execução em um arquivo de nome logTempo.db e em seguida os imprimindo em tela.

Todos os valores são iniciais e sendo "Inserção"o tempo em milissegundos para os registros serem inseridos e "Busca"o tempo de busca e impressão dos valores.

Tabela 1 – Testes iniciais

k	p	n	m	Inserção (ms)	Busca (ms)
1000	8	1000	1000	12760	8657
10000	8	1000	1000	226959	38117

Sendo assim os teste passaram a ser feitos utilizando 1000 registros, com diferentes valores de p, n e m de forma a tornar possível a visualização do efeito de diferentes valores sobre os tempos de inserção e busca. Utilizando de um número menor de registros, foi possível um tempo de resposta ágil, porém, ainda possível de visualizar o impacto que cada variável tem no programa.

3.3 Testes

Todos os testes foram executados 3 vezes e os valores de inserção e busca são a média dos testes individuais.

3.3.1 SSD

Os resultados dos testes indicados na tabela abaixo foram todos executados em um SSD.

Tabela 2 – Testes em SSD

Teste	k	p	n	m	Inserção (ms)	Busca (ms)
1	1000	3	100	100	2856	1045
2	1000	3	100	1000	3107	1692
3	1000	3	500	100	10244	1288
4	1000	5	100	100	2902	1066
5	1000	5	100	1000	3160	1838
6	1000	5	500	100	10675	1593
7	1000	8	100	100	3275	1950
8	1000	8	100	1000	2974	2637
9	1000	8	500	100	10673	4451
10	1000	12	100	100	3361	16905
11	1000	12	100	1000	3040	17201
12	1000	12	500	100	11229	54350

3.3.2 HD Externo

Os resultados dos testes indicados na tabela abaixo foram todos executados em um HD Externo utilizando USB 2.0.

Tabela 3 – Testes em HD

Teste	k	p	n	m	Inserção (ms)	Busca (ms)
1	1000	3	100	100	4576	949
2	1000	3	100	1000	4654	1608
3	1000	3	500	100	14055	1263
4	1000	5	100	100	4490	1004
5	1000	5	100	1000	4729	1690
6	1000	5	500	100	14734	1665
7	1000	8	100	100	4681	2097
8	1000	8	100	1000	4775	2711
9	1000	8	500	100	14965	5856
10	1000	12	100	100	4487	19748
11	1000	12	100	1000	4824	20518
12	1000	12	500	100	14551	76777

3.4 Análise de resultados

Quando comparados os valores entre os diferentes testes, torna-se claro que o número de entradas por *bucket* (n) é a variável de maior impacto para a inserção de registros. Em alguns casos o incremento desta aumenta o tempo de inserção em mais de três vezes o valor quando comparado com o base. No entanto, o tamanho das anotações do registro (m) afetou o tempo de busca e impressão nos teste iniciais, porém, em profundidades globais mais altas, o valor de n foi o de maior impacto para ambos os tempos de execução.

Comparando os testes entre SSD e HD externo, os resultados foram bastante similares em relação ao seu crescimento e variação de acordo com os parâmetros. A principal diferença é que em todos os testes os tempos de inserção foram maiores quando testados no HD externo, demonstrando assim o impacto que hardwares diferentes podem ter quanto a escrita em disco. No HD os tempos de busca e impressão foram menores em valores de p mais baixos, o mesmo ocorreu em todas as repetições, indicando não ser uma falha no teste. Porém, em valores de p mais elevados o SSD se demonstrou mais uma vez mais rápido, obtendo tempos menores que o HD.

Figura 1 – Gráfico comparativo de tempos de inserção

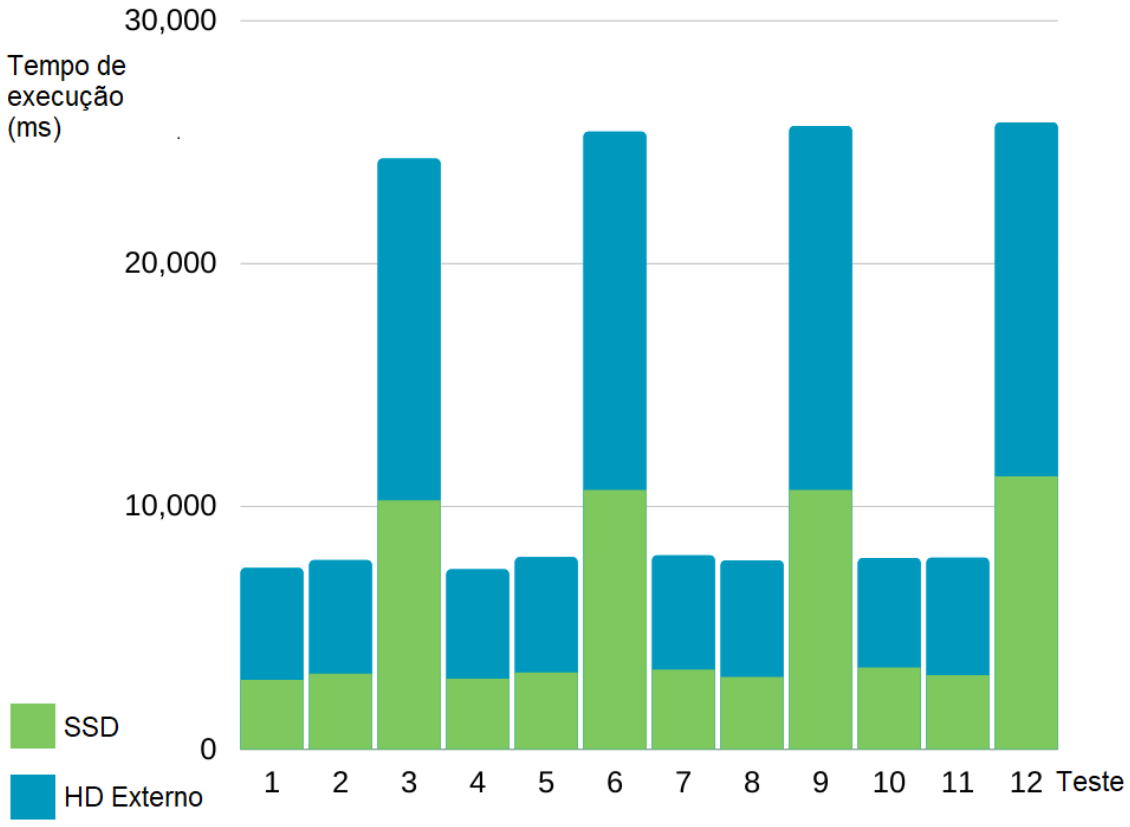
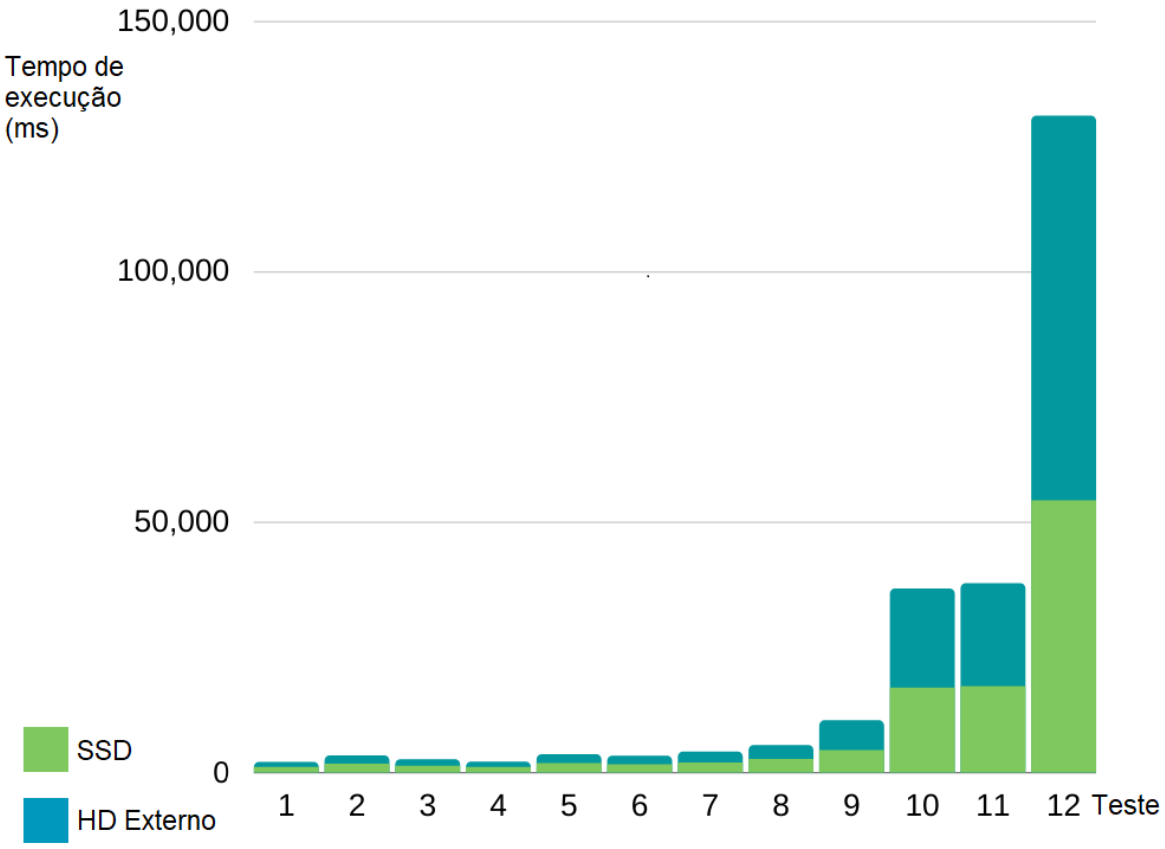


Figura 2 – Gráfico comparativo de tempos de busca



4 CONCLUSÃO

O funcionamento do hash extensível é muito interessante por ao mesmo tempo ser um conceito simples mas com uma implementação bem complexa. Ao longo do desenvolvimento deste trabalho ficou bem claro esta complexidade, exigindo diversas fontes e conhecimentos, de forma a ser possível o entendimento do que seria necessário para a execução dele. Ressalta-se que este trabalho foi feito individualmente e, portanto, não foi possível a implementação correta e satisfatória no tempo estipulado, devido à sua complexidade e grandeza.

Devido a limitação de hardware, os testes foram executados em uma escala muito menor do que o ideal. Porém, ainda assim, os resultados indicam a grande diferença que a alteração dos parâmetros iniciais podem ter sobre o tempo de execução do programa, tanto na inserção dos prontuários médicos, quanto na busca e impressão destes. Desta forma, demonstra a efetividade do hash extensível para grandes quantidades de registros, reduzindo muito os tempos de execução quando comparados com outros métodos de busca, como por exemplo, a busca linear, tendo porém, um menor impacto em quantidades menores de registros.

Acredito que seja possível uma ótima implementação do hash extensível em um tempo mais longo devido a ele ser conceitualmente simples e pela linguagem utilizada (Java) ser de grande auxílio na execução e aplicação deste método de organização de registros.

5 BIBLIOGRAFIA

VIDEOAULAS da discilpina Algoritmos e Estruturas de Dados III. Gravação de Zenilton Kleber Gonçalves do Patrocínio Júnior e Marcos André Silveira Kutova. [S. l.]: PUC Minas, 2021. Disponível em: https://pucminas.instructure.com/courses/82829. Acesso em: 3 dez. 2021.

EXTENDIBLE Hashing: Dynamic approach to DBMS. [S. l.], 22 mar. 2021. Disponível em: https://www.geeksforgeeks.org/extendible-hashing-dynamic-approach-to-dbms/?ref=gcse. Acesso em: 21 nov. 2021.

TABELAS de dispersão (hash tables). [S. l.]: Paulo Feofiloff, 9 out. 2017. Disponível em: https://www.ime.usp.br/ pf/mac0122-2002/aulas/hashing.html. Acesso em: 19 nov. 2021.

LITWIN, Witold. Linear hashing: a new tool for file and table addressing. In Proceedings of the sixth international conference on Very Large Data Bases - Volume 6 (VLDB '80). VLDB Endowment, p. 212–223, 1980.

FAGIN, Ronald et al. Extendible hashing—a fast access method for dynamic files. ACM Transactions on Database Systems (TODS), v. 4, n. 3, p. 315-344, 1979.