

Relatório do Trabalho Prático

Algoritmos e Estruturas de Dados III

Alexandre de Carvalho Jurka¹, Vinícius Tavares Coimbra Ribeiro²

¹Instituto de Ciências Exatas e Informática
Pontifícia Universidade Católica de Minas Gerais (PUC-MG)
30.535-901 – Belo Horizonte – MG – Brasil

Resumo. *Este trabalho aborda diversas técnicas de processamento de dados no contexto de memória secundária. É abordado a conversão de formatos de arquivo .csv para .bin, explorando indexação de registros através de árvore B+, hash extensível e lista invertida. Destaque para a diferença entre estratégias de busca sequencial e indexada, além de analisar algoritmos de compressão (LZW, Huffman) para redução de tamanho de dados. O trabalho inclui uma análise de algoritmos de casamento de padrão (KMP, Boyer Moore) para identificação eficaz de padrões. Também implementa o algoritmo de criptografia RSA, oferecendo insights sobre sua aplicação na proteção de dados sensíveis.*

1. Introdução

Este trabalho apresenta a implementação prática, na linguagem Java, de diversos algoritmos e estruturas de dados para comparar o processamento e manipulação de informações em memória secundária.

Inicia-se com a eficiente conversão de formatos de arquivo, demonstrando a transformação de arquivos .csv (base de dados escolhida) para .bin (binário). A base de dados escolhida foi uma base de animes com os campos: **Name, Type, Episodes, Studio, Tags, Rating, Release_year**.

A indexação de registros é abordada através de árvore B+, hash extensível e lista invertida, explorando suas características para otimizar a busca e recuperação de dados. Estratégias de busca sequencial e indexada são implementadas, proporcionando uma análise comparativa de desempenho em cenários diversos.

O trabalho inclui a implementação e análise de algoritmos de compressão, como LZW e Huffman, destacando sua eficácia na redução do tamanho de conjuntos de dados. Algoritmos de casamento de padrão, como KMP e Boyer Moore, são implementados para identificação eficiente de padrões em grandes conjuntos de dados.

Finalmente, o algoritmo de criptografia RSA é implementado, oferecendo uma camada adicional de segurança na proteção de dados sensíveis. A pesquisa abrange aspectos práticos de implementação, segurança e desempenho, proporcionando uma visão abrangente do papel desses algoritmos em diversas aplicações.

Em conjunto, este trabalho não apenas explora teoricamente, mas também coloca em prática uma gama diversificada de algoritmos e estruturas de dados, destacando sua relevância e eficácia na manipulação eficiente de dados em memória secundária.

2. Desenvolvimento

2.1. Organização do código

Um diretório com o nome de *model* foi utilizado para armazenar os códigos que modelam cada uma das estrutura de dados utilizadas. No diretório *dao* foram armazenados os códigos que manipulam os dados na memória externa diretamente, seja para ler ou escrever dados. O diretório *util* contém os códigos que possuem alguma utilidade não relacionada com o processo de maneira essencial e sim para melhorar a experiência de navegação ou organização de informação.

2.1.1. model

- Anime
- Arquivo
- BitManipulation
- Bits
- BPlusTreePage
- DataBucket
- DataRecords
- DynamicHashing
- Huffman
- ListaInvertida
- LZW
- PageElement
- Record
- Tape
- RSA

2.1.2. dao

- AnimeDAO
- BitManipulationDAO
- BoyerMooreDAO
- BPlusTreeDAO
- DynamicHashingDAO
- HuffmanDAO
- KMPDAO
- ListaInvertidaDAO
- LZWDao
- RecordDAO
- RSADao

2.1.3. util

- ProgressMonitor
- Timer

2.1.4. Resources

A pasta *resources* contém o arquivo original .csv, além dos arquivos resultantes das etapas de conversão para binário, indexação, etc. Dessa forma, a pasta *resources* é o destino de qualquer arquivo gerado pelo código durante sua execução.

2.2. Estruturação dos Dados

2.2.1. AnimeDAO

Os registros são manipulados pela classe *AnimeDAO*. Essa classe possui todos os métodos que manipulam os registros, por exemplo: conversão para binário, compressão do arquivo, casamento de padrão, etc.

Os registros seguem o seguinte modelo:

- **tamanho**: inteiro de 4 bytes.
- **id**: inteiro de 4 bytes.
- **nome**: texto de tamanho variado de caracteres.
- **tipo**: texto de tamanho fixo de 5 caracteres.
- **episódios**: inteiro de 4 bytes.
- **estúdio**: texto de tamanho variado de caracteres.
- **tags**: texto de tamanho variado de caracteres e valores separados por vírgulas.
- **rating**: ponto flutuante de 4 bytes.
- **lançamento**: timestamp de 8 bytes.

Uma observação importante sobre o **tamanho** do registro é que nesses 4 bytes estão armazenadas duas informações: a **lápide** do registro e seu **tamanho**. O bit de sinal desse inteiro (bit mais significativo) é a lápide e o resto o tamanho em si. Isso implica na economia de 1 byte por registro na base de dados, mas com o contrapeso de reduzir a capacidade de representações do tamanho do registro pela metade. Julgando os prós e contras, a decisão de usar a lápide de um bit foi incorporada ao projeto com fácil implementação.

2.2.2. Página

A estrutura da página usada nos arquivos foi montada de acordo com cada algoritmo, mas todas as páginas possuem os *elementos da página*, que são estruturados da seguinte forma:

- **id**: inteiro de 4 bytes que contém o id do registro.
- **pointer**: inteiro de 8 bytes que contém o endereço no arquivo de dados do registro.

2.2.3. Indexação

A indexação do arquivo foi feita com as estruturas de **Árvore B+**, **Lista Invertida** e **Hashing Flexível**.

- **Árvore B+**

Na estrutura da **Árvore B+** há uma pequena otimização implementada. Como o endereço do registro no arquivo de dados está armazenado apenas nas

folhas e as páginas internas contêm apenas duplicatas, foi utilizado um tamanho fixo para as páginas com o âmbito de utilizar o espaço inutilizado do endereço, nas páginas internas, para armazenar o endereço da página folha da estrutura. Assim, quando encontrar o registro em uma página interna, podemos ir direto para a folha em que está armazenada poupando a necessidade de percorrer outras páginas resultando assim em menos leituras em memória externa.

- Hash Flexível

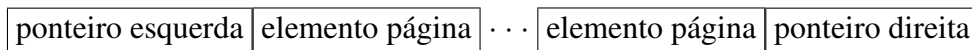
Na estrutura de Hash Flexível foi utilizado o tamanho padrão de 975 elementos no *bucket* (5% do total de registros).

Nenhuma otimização foi utilizada, além de juntar *buckets* semi-vazios para diminuir o tamanho do arquivo de indexação.

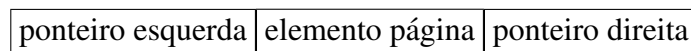
- Páginas

A estrutura da página depende de cada estrutura de indexação. Como temos a estrutura de elemento da página, podemos montar as estruturas em cada DAO (data access object) de maneira específica.

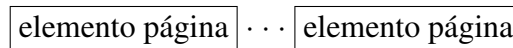
No caso da Árvore B+, como a ordem pode ser escolhida sob demanda, a estrutura básica é:



entre esse modelo (\dots) pode ter $N \leq \infty$ da seguinte estrutura:



A estrutura no Hash Dinâmico se dispõe da seguinte forma:



a quantidade de **elementos página** depende do tamanho do *bucket*.

2.2.4. Manipulação em nível de bit

A manipulação em nível de bit -usado na compressão, por exemplo- é realizada pelas classes BitManipulationDAO e pela classe Bits. Essas classes escrevem e leem informação de bit a bit, o que torna a manipulação mais precisa ao custo de desempenho por causar mais acessos a memória.

2.2.5. Pesquisa de Registros

A pesquisa de um registro é feita de maneira sequencial e indexada, utilizando as estruturas de indexação previamente mencionadas. Essa pesquisa é utilizada como apoio para algumas operações de CRUD.

2.2.6. CRUD

As operações de CRUD alteram o arquivo de dados. Dessa forma há a possibilidade que ocorra alterações necessárias nos arquivos de indexação. Tais ocorrências são previstas

e realizam mudanças nos arquivos de indexação para que suas informações continuem a condizer com as do arquivo de dados. O *create*, por exemplo, também insere o **ID** e o **endereço no arquivo de dados** do novo elemento cadastrado. Assim, a utilização da pesquisa continua sendo válida.

2.2.7. Casamento de Padrões

Os casamentos de padrões(*KMP* e *Boyer Moore*) são realizados diretamente no arquivo de dados. Ambos algoritmos proporcionam a quantidade de comparações realizadas para encontrar o padrão.

2.2.8. Criptografia

O algoritmo de criptografia(*RSA*) é usado para realizar a criptografia e descriptografia do arquivo de dados todas as vezes que seja necessário a manipulação do mesmo, seja para um simples mostrar de todos registros até uma ordenação dos registros.

2.3. Navegação do Usuário

A navegação pelo aplicativo foi feita através de uma interface gráfica no próprio terminal de execução. Alguns recursos- como o *Progress Monitor*, mencionado anteriormente- foram criados a fim de proporcionar uma experiência mais amigável e compreensiva para o usuário, afim de ser mais claro e transparente sobre os processos que estão ocorrendo e seu progresso, seja a porcentagem atual ou o tempo gasto para sua completude.

2.4. Notas Adicionais

Alguns tópicos foram feitos puramente com o intuito de otimizar, refinar ou embelezar o projeto foram feitos. Tais tópicos são:

- Lápide de um bit

A lápide dos registros gasta apenas um bit de espaço. Esse bit é o bit de sinal, ou bit mais significativo, do tamanho do registro, que é um inteiro de 4 bytes. Essa otimização foi feita para economizar um byte por cada registro no arquivo de dados.

A alteração desse bit não pode interferir nos outros bits do tamanho do registro. Para que isso ocorra, a alteração da lápide se dá através de uma operação de **xor**(\oplus) entre o byte mais significativo do tamanho do registro com o byte: *'0b10000000'*. A operação: $0 \oplus x = x$ preserva o valor do bit x , mas a operação: $1 \oplus x = \bar{x}$. Assim, podemos negar o bit de sinal sem alterar os demais bits do tamanho do registro ao usarmos *'0b10000000'*.

Para verificar se um registro é válido ou não, basta verificar se seu tamanho é negativo ou não. Caso seja positivo, não há nada a mais para ser feito. Caso seja negativo, para recuperar seu tamanho é necessário negar o bit de sinal apenas.

- Multithread

O *Progress Monitor*, inicialmente, foi implementado na mesma *thread* de execução do programa principal. Isso acarretou em um aumento considerável no tempo de execução dos módulos, que em certos casos chegou a ser quase $7x$ mais

lento que o mesmo módulo sem a utilização dessa funcionalidade.

Para resolver esse problema, o uso de múltiplas *threads* foi adotado. O tempo de execução do programa principal foi afetado de forma que seu aumento, ao usarmos o *Progress Monitor*, foi considerado insignificante em pró do valor que essa funcionalidade agrega ao projeto.

- GUI

Uma pasta no projeto chamada **menu** contem o desenvolvimento de uma GUI (*Graphical User Interface* ou *Interface Gráfica do Usuário*) usando a biblioteca *java.swing*

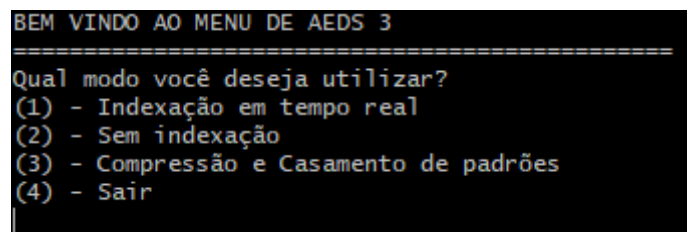
Essa interface funciona, e faz, parcialmente, o que a navegação pelo terminal faz. Terminar e refinar essa interface é um objetivo que será feito no futuro, mas como não faz parte da entrega e sim algo extra, não recebeu prioridade sobre as funcionalidades do sistema.

Essa GUI utiliza **multithreading**. A interface em si será executada na *thread* principal e os processos a serem realizados- como a conversão de csv para binário, por exemplo- serão feitos em outra *thread*. O processo do *java.swing.ProgressMonitor* também é feito em uma *thread* própria.

Caso algum desses processos destacados acima não seja feito em *threads* separadas, a interface gráfica não funcionará corretamente.

3. Testes e Resultados

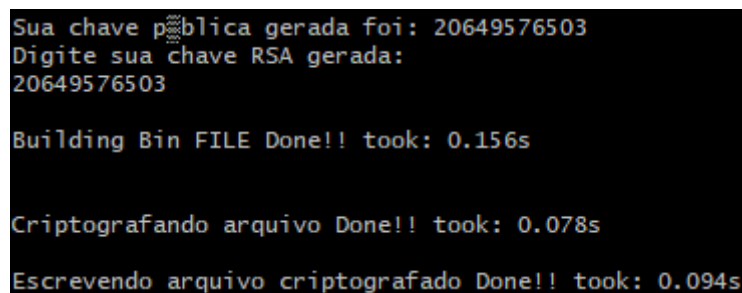
O fluxo do programa começa com um menu inicial. Uma criptografia inicial é feita e, além



```
BEM VINDO AO MENU DE AEDS 3
=====
Qual modo você deseja utilizar?
(1) - Indexação em tempo real
(2) - Sem indexação
(3) - Compressão e Casamento de padrões
(4) - Sair
```

Figure 1. Menu principal

disso, a cada manipulação do arquivo de dados ocorre a descriptografia e criptografia do mesmo, por exemplo, no momento de criação da Árvore B+. Isso acaba adicionando mais



```
Sua chave pública gerada foi: 20649576503
Digite sua chave RSA gerada:
20649576503

Building Bin FILE Done!! took: 0.156s

Criptografando arquivo Done!! took: 0.078s

Escrevendo arquivo criptografado Done!! took: 0.094s
```

Figure 2. Criptografia Inicial

processamento, ou seja, aumenta o tempo de execução de cada opção.

Após alguns processos iniciais serem realizados um menu principal é mostrado e usado

```

Digite a ordem da árvore:
8
Files successfully created!!

Descriptografando arquivo Done!! took: 0.125s

Building B+ Tree Done!! took: 1.081s

Criptografando arquivo Done!! took: 0.203s

Escrevendo arquivo criptografado Done!! took: 0.749s

```

Figure 3. Descriptografia e Criptografia

```

0 que deseja fazer?
(1) - Buscar Anime
(2) - Listar Animes
(3) - Apagar Anime
(4) - Inserir Anime
(5) - Atualizar Anime
(6) - Buscar animes por tipo
(7) - Buscar anime por estúdio
(8) - Ordenar Animes
(9) - Indexar Animes
(0) - Sair

```

Figure 4. Menu Principal

no decorrer da execução. Ao escolher pesquisa é necessário digitar o **ID** a ser buscado. Logo em seguida será possível escolher entre Árvore B+ ou Hash como estruturas da busca, caso a opção de indexação em tempo real tenha sido escolhida, caso contrário será feita uma busca sequencial.

```

Digite o ID buscado
386
=====
Digite em qual tipo de index você deseja utilizar
(1) - Arvore B+
(2) - Hash Extensível
=====
2
Searching id: 386 Done!! took: 1.062s

```

Figure 5. Pesquisa

Ao escolher a opção de *Compressão e Casamento de Padrão* um submenu é mostrado. Os casamentos de padrão são feitos no arquivo csv original e após concluir a busca do padrão é mostrado a quantidade de comparações realizadas. O algoritmo **KMP** mostra a função de erro do padrão digitado além das coisas mencionadas acima. No quesito desempenho o algoritmo *Boyer Moore* demonstra maior eficiência no número de comparações se comparado ao *KMP*.

```

O que deseja fazer?
(1) - Buscar padrão por KMP
(2) - Buscar padrão por Boyer Moore
(3) - Comprimir com Huffman
(4) - Comprimir com LZW
(5) - Sair

```

Figure 6. Menu de Compressão e Casamento

```

Digite o padrão que deseja buscar:
SeasonSeason
Carregando arquivo csv
Leitura finalizada

KMP Pattern Matching Done!! took: 0.013s
O número de comparações foi: 2224292
Função de falha:
Failure Function (SeasonSeason):
  j: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
f(j): | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

```

Figure 7. Algoritmo KMP

Os algoritmos de compressão mostram as etapas que estão sendo realizadas do processo e, após sua conclusão, também mostra o quanto foi comprimido em relação ao arquivo original. O algoritmo de compressão *LZW* obteve a melhor taxa de compressão

```

Preparing File Done!! took: 4.349s

Building Dictionary Done!! took: 0.032s

Building Huffman Tree Done!! took: 0.0s

Compressing Done!! took: 0.156s
Completed!! COMPRESSION Rate: 35,68%
Decompressing File Done!! took: 0.094s

```

Figure 8. Huffman

```

Compression rate: 70,52%

Reading compressed File Done!! took: 10.978s
Decompressing Done!! took: 0.218s

```

Figure 9. LZW

se comparado a *Huffman*. Contudo o tempo gasto no geral pelo *Huffman* é consideravelmente menor que o tempo gasto pelo *LZW*.

*obs.: Para maiores detalhes sobre os testes e resultados, além de uma demonstração mais profunda do fluxo de execução, recomendamos assistir os vídeos:

<https://www.youtube.com/watch?v=ZLcld40tG6Q>

<https://www.youtube.com/watch?v=Lz6qAuS4ZI>

4. Conclusão

A conclusão deste trabalho revelou uma compreensão mais aprofundada e minuciosa do processo de manipulação de arquivos externos. Através da implementação de algoritmos responsáveis por essa manipulação, foi possível adquirir conhecimentos valiosos sobre as limitações amarradas à máquina ao lidar com grandes volumes de dados. Para enfrentar essas restrições, exploramos diversas abordagens, cada uma apresentando suas próprias vantagens e desvantagens.

A experiência proporcionou um aprendizado significativo, destacando a importância fundamental de entender as nuances envolvidas na manipulação de dados em programação. Esse conhecimento adquirido não apenas contribui para uma prática mais refinada, mas também promove a robustez e a segurança no desenvolvimento de programas. Aprofundar-se no fluxo de dados e suas manipulações não apenas amplia a capacidade técnica, mas também oferece uma perspectiva mais ampla sobre as escolhas de design e a otimização de algoritmos.

Assim, a conclusão deste trabalho não apenas ampliou nossa compreensão prática da manipulação de arquivos externos, mas também forneceu uma base sólida para abordagens mais sofisticadas e eficientes na programação. Essas percepções são inestimáveis para aqueles que buscam não apenas desenvolver soluções funcionais, mas também aprimorar suas habilidades em direção a uma programação mais eficaz e adaptável.