# Relatório de Implementação Algoritmos e Estrutura de Dados III

**Allan G. Gomes Pego**[1], **André Santos Alves** [2]

[1]Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais (PUC-MG)
Belo Horizonte – Minas Gerais – Brazil

***Resumo.*** *Este relatório documenta as soluções implementadas para os trabalhos práticos da disciplina de Algoritmos e Estruturas de Dados III (AEDS III) no semestre 2023/2, cujo principal enfoque foi a manipulação de dados na memória secundária. O documento detalhará os métodos, as escolhas de projeto e os resultados comparativos de cada implementação realizada nos trabalhos práticos 1, 2 e 3.*

## 1. Introdução

A memória secundária desempenha um papel fundamental no funcionamento dos computadores modernos, proporcionando a possibilidade de armazenar dados a longo prazo e em grande quantidade, o que a diferencia da memória principal, caracterizada por menor capacidade de armazenamento e volatilidade dos dados ali armazenados.

No entanto, esses benefícios são acompanhados de um custo - a leitura e a escrita de dados na memória secundária são consideravelmente mais lentas do que os mesmos processos feitos em memória primária -, o que torna o fluxo de dados entre a Unidade de Processamento Central (CPU) e a memória secundária um grande gargalo para aplicações que dependem dele. Nesse sentido, é proposto na disciplina de AEDS III e nos trabalhos práticos aqui documentados o estudo de estruturas de dados e algoritmos com potencial de maximizar a eficiência das operações com manipulação de dados em memória secundária.

O objetivo individual de cada TP está representado na listagem abaixo.

TP1: Transformação de arquivos de dados em arquivos binários (.db), operações CRUD (Create, Read, Update e Delete), ordenação externa e indexação e CRUD indexado em arquivos binários.

TP2: Compactação de arquivos e casamento de padrões.

TP3: Criptografia.

## 2. Desenvolvimento

### 2.1. TP1 - Basse de dados com Arquivos .db

### 2.2. Criação do Arquivo .db

Foi escolhida a base de dados de Jogadores de ”Counter Strike: Go”, disponibilizada pela plataforma Kaggle, para montar a base de dados do TP. Essa base foi extraída em um arquivo CSV, tratada e convertida em um arquivo .db. A entidade jogador, representada

em Java pela classe Player, foi construida com os seguintes campos: ID (inteiro), nickname (string tamanho variável), data de nascimento (gerada aleatoriamente a partir do ano de nascimento disponível na base CSV), lista de times (string separada por vírgulas), e nacionalidade (string de tamanho fixo, padrão ISO 3166-1 alfa-3). Já o registro, o PlayerRegister em Java, possui como atributos um indicador booleano de lápide, um inteiro indicador do tamanho do registro e um jogador. Cada uma das classes foi implementada no package model e possui métodos que permitem a leitura e escrita em vetor de bytes, necessário para construção do arquivo .db. Com essas implementações, foram criados as classes do package fileHandler em Java, responsáveis por fazer a leitura e escrita dos dados presentes tanto no CSV, quando nos arquivos .db.

Além da estrutura dos registros mencionada anteriormente, o arquivo binário possui um cabeçalho, constituído de seus quatro primeiros bytes, que armazena o maior ID encontrado na base, o que auxilia nas operações de CRUD.

### 2.2.1. CRUD Sequencial e Indexado

Implementação da lógica de inserção (Create), leitura (Read), atualização (Update), e deleção (Delete) de dados na base de dados .db e, caso essa base de dados se encontre indexada, nos arquivos de índice. No código Java, as lógicas estão presentes nas classes do package DAO.

As operações CRUD possíveis estão detalhadas abaixo:

- Ler um registro (id): esse método recebe um id como parâmetro, percorre o arquivo binário e retorna os dados do id informado.
- Atualizar um registro: esse método recebe novas informações sobre um objeto e atualiza os valores dele no arquivo binário. Se o novo registro possuir tamanho menor ou igual ao anterior os dados são alterados no próprio local, e o tamanho do novo registro mantém o do antigo. Porém, caso o novo registro seja maior do que o anterior, o local é marcado como lápide e o novo registro é escrito ao final do arquivo.
- Deletar um registro (id): esse método recebe um id como parâmetro, percorre o arquivo binário e coloca uma marcação (lápide) no registro que será considerado deletado.

### 2.2.2. Ordenação Memória Secundaria

Dado o tamanho que os arquivos em memória secundária podem ter, faz-se necessária a implementação de lógicas de ordenação que não necessitem de todas as informações serem trazidas para memória principal. Nesse sentido, foram utilizados três métodos:

1. Intercalação balanceada comum
2. Intercalação balanceada com blocos de tamanho variável
3. Intercalação balanceada com seleção por substituição

Cada método permitia a entrada de usuário da quantidade de registros que poderiam ser trazidos para memória primária, além de ter parametrizado a quantidade de arquivos que seriam utilizados no balanceamento da ordenação.

Para garantir integridade com operações de CRUD indexado, quando o usuário sai da aba de ordenação, o programa automaticamente cria uma nova indexação para os registros ordenados.

### 2.2.3. Indexação

A indexação é fundamental para permitir o acesso aos dados sem precisar percorrer sequencialmente os registros da base original, que costumam ter tamanho consideravelmente maior do que cada subconjunto dos arquivos indexados. Para essa tarefa, utilizamos Hash Estendido e Árvore B. As duas indexações foram realizadas utilizando o id dos registros.

O Hash utilizou a função h(id) = id mod $2^p$, em que p é a profundidade do diretório, inicializada em zero e que aumenta a cada bucket, de tamanho parametrizável em até 5% o tamanho da base, que é totalmente preenchido.

A indexação por hash conseguiu refletir as 4 operações de CRUD através do CRUD indexado. A indexação utilizando Árvore B possui ordem 8 e reflete as operações de inserção, leitura e update do CRUD, apenas.

## 2.3. TP2 - Compressão de Arquivos e Casamento de Padrões

### 2.3.1. Compactação

A compactação dos dados é de grande importância para maximizar o aproveitamento da memória secundária e em outros contextos em que a quantidade de informação é deve ser limitada.

Para compactar o arquivo binário, utilizamos os algoritmos LZW e Huffman. Em nossa implementação, a compressão de Huffman consiste em diminuir a quantidade de bits utilizada para representar os caracteres mais utilizados. Em nossa implementação, essa compressão foi feita byte a byte, passando duas vezes pelo arquivo, uma para contar as repetições de caracteres e outra para escrever as novas representações de cada byte no arquivo de saída.

O LZW constiste em tentar reduzir o tamanho do arquivo ao utilizar uma única representação para um conjunto de caracteres. Em nossa implementação, esse algoritmo faz uma compressão dos campos do registros após os transformar em string (array de caracteres ou bytes). O resultado dessa compressão é sempre um array de doubles (ou seja, um array de dois bytes). A ideia foi aumentar o tamanho do dicionário LZW de 128 palavras (o tamanho unsigned de um byte) para 32768 (o tamanho unsigned de um double) para aumentar o número de repetições que podiam ser compactadas em um símbolo apenas. Em outras palavras, o grupo tomou a decisão arrojada de tentar aumentar a taxa de compressão aumentando o tamanho da representação dos símbolos do dicionário, o que poderia ter efeito contrário caso o número de repetições de conjuntos de caracteres no arquivo original fosse baixo. Apesar do risco que se escolheu tomar, os resultados foram favoráveis à escolha.

### 2.3.2. Casamento de Padrões

O casamento de padrões é uma etapa crucial em muitos algoritmos e aplicações, e para essa tarefa, empregamos os eficientes métodos de Boyer-Moore e Knuth-Morris-Pratt (KMP), aplicando-os diretamente aos registros convertidos em strings.

O algoritmo de Boyer-Moore destaca-se pela sua capacidade de realizar casamento de padrões de maneira eficiente, utilizando estratégias como o salto de caracteres e a tabela de saltos para minimizar o número de comparações necessárias. Isso torna o Boyer-Moore especialmente útil para lidar com grandes conjuntos de dados.

Por outro lado, o algoritmo KMP, baseado na ideia de construir uma tabela de prefixo que permite evitar comparações redundantes, é uma escolha eficaz para padrões que contêm prefixos repetidos. Essa abordagem oferece um desempenho consistente, especialmente em cenários nos quais a repetição de prefixos é comum.

### 2.4. TP3 - Criptografia

Para criptografar os dados, utilizamos as técnicas de Vigenere, na qual somamos os characteres das strings de entrada com os da string chave, que é percorrida de forma circular; e cifra das colunas, que troca a ordem do texto de acordo com a ordem alfabética da chave.

## 3. Testes e Resultados

### 3.1. TP1 - Basse de dados com Arquivos .db

Para o TP1, exceto a deleção da Árvore B, todos os métodos de CRUD, Indexação, CRUD Indexado, Ordenação Externa e CRUD Indexado após ordenação externa funcionaram como esperado e corretamente, sem quaisquer gargalos.

### 3.2. TP2 - Compressão de Dados e Casamento de Padrões

No que tange compressão de dados, os algoritmos LZW e Huffman conseguiram compactar e descompactar corretamente o arquivo original. Os dois métodos conseguiram transformar o tamanho do arquivo original, 48.8KB, para o mesmo valor de 35.62KB, uma compressão de 27.05%. O Algoritmo de Huffman foi mais eficiente para diminuir o tamanho do arquivo, já que utilizou 2 bytes a menos que LZW. Entretanto, o algoritmo LZW foi mais rápido tanto na compressão quanto na descompressão, já que gastou 120ms e 83ms em cada uma das operações, contra 317ms e 115ms do algoritmo de Huffman em nosso ambiente de testes.

Já em relação ao casamento de padrões, os algoritmos Boyer Moore e KMP conseguiram realizar corretamente a tarefa. O algoritimo Boyer Moore apresentou maior eficácia em nossos testes, ao procurar por ”fallen”, um padrão conhecido e presente na base de dados, o algoritmo necessitou de 3.603 comparações, contra 19.831 do KMP. Já em uma situação que os algoritmos precisam percorrer toda a base, o algoritmo Boyer Moore também foi mais efetivo, realizando 25.500 comparações, contra 92.408 do KMP.

### 3.3. TP3 - Criptografia

Enquanto a criptografia, foram escolhidos dois algoritmos simples, mas que conseguem criptografar e descriptografar corretamente os arquivos.

## 4. Conclusão

Ao finalizar os três trabalhos práticos, podemos concluir que obtivemos implementações eficazes e consistentes, que nos permitiram extrair conhecimento tanto de implementação, quanto da aplicação de cada método desenvolvido. Por exemplo, na compactação, o LZW é adequado em contextos em que há necessidade de se fazer compactações e descompactações rápidas, principalmente pelo fato de não precisar ler o arquivo de origem duas vezes, enquanto o Huffman pode ser mais adequado quando a necessidade é de se reduzir a quantidade de dados. Já na busca por padrões o algoritmo de Boyer Moore tem capacidade de reduzir o número de buscas em memória secundária em relação ao KMP.