Sistema de Gerenciamento de Base de Dados de Animes

Estruturas de Dados, Compressão e Criptografia

Trabalhos Práticos I, II, III e IV
Algoritmos e Estruturas de Dados III

Autores:

Júlia de Mello

Alex Marques

Pontifícia Universidade Católica de Minas Gerais
Curso de Engenharia de Computação

26 de junho de 2025

Sumário

1 RESUMO

Este relatório documenta o desenvolvimento de um sistema completo de gerenciamento de dados de animes, implementado ao longo de quatro trabalhos práticos da disciplina de Algoritmos e Estruturas de Dados III. O sistema evolui desde uma implementação básica com operações CRUD até um sistema robusto com múltiplos tipos de índices, algoritmos de compressão e técnicas de criptografia. O projeto demonstra a aplicação prática de estruturas de dados avançadas como Árvore B+, Hash Extensível e Lista Invertida, além de algoritmos de compressão (Huffman e LZW), casamento de padrões (Boyer-Moore e KMP) e criptografia (Vigenère e RSA). Os resultados mostram significativa melhoria na eficiência das operações de busca e considerável redução no tamanho dos arquivos através da compressão, enquanto a criptografia garante a segurança dos dados armazenados.

2 INTRODUÇÃO

O gerenciamento eficiente de grandes volumes de dados é um desafio fundamental na ciência da computação. Este projeto implementa um sistema completo para gerenciamento de uma base de dados de animes, explorando diversas técnicas e estruturas de dados para otimizar operações de armazenamento, busca, compressão e segurança.

O desenvolvimento foi dividido em quatro etapas incrementais:

- **TP1**: Implementação básica com CRUD e ordenação externa
- **TP2**: Adição de estruturas de indexação (Árvore B+, Hash Extensível, Lista Invertida)
- **TP3**: Implementação de compressão (Huffman, LZW) e casamento de padrões (Boyer-Moore, KMP)
- **TP4**: Adição de técnicas de criptografia (Vigenère, RSA)

O sistema utiliza Java como linguagem de implementação, manipulando arquivos binários para armazenamento persistente e oferecendo uma interface de linha de comando para interação com o usuário.

3 DESENVOLVIMENTO

3.1 TP1 - Fundamentos e Operações Básicas

3.1.1 Estrutura do Arquivo Principal

O arquivo binário principal (`animeDataBase.db`) foi projetado com a seguinte estrutura:

- **Cabeçalho**: 4 bytes contendo o número total de registros (18.495)
- **Registros**: Cada registro contém:
  - Lápide (2 bytes): Marca se o registro está ativo (' ') ou excluído ('*')
  - Tamanho (2 bytes): Tamanho do registro em bytes
  - Dados serializados: Objeto Anime convertido em bytes

3.1.2 Classe Animes

A classe principal representa um anime com os seguintes atributos:

- `id`: Identificador único
- `name`: Nome do anime
- `type`: Tipo de mídia (5 bytes fixos)
- `episodes`: Número de episódios
- `rating`: Nota do anime
- `year`: Data de lançamento (classe MyDate)
- `genres`: Lista de gêneros
- `season`: Temporada de lançamento
- `studio`: Estúdio responsável

3.1.3 Operações CRUD

Implementou-se as quatro operações básicas:

- **Create**: Inserção de novos registros no final do arquivo
- **Read**: Busca sequencial por ID
- **Update**: Atualização in-place quando possível, realocação caso contrário
- **Delete**: Exclusão lógica marcando a lápide

3.1.4 Ordenação Externa

Foi implementada uma classe de ordenação externa para organizar o arquivo após modificações. O algoritmo utiliza:

- **Distribuição**: Divisão do arquivo em sessões menores
- **Ordenação interna**: Algoritmo de seleção para cada sessão
- **Intercalação**: Merge das sessões ordenadas

Nota: Esta classe foi posteriormente descontinuada com a implementação dos índices no TP2, pois a indexação tornou desnecessária a manutenção de ordem física dos registros.

3.2 TP2 - Estruturas de Indexação

A adição de índices revolucionou a eficiência do sistema, eliminando a necessidade de busca sequencial e ordenação externa.

3.2.1 Árvore B+

Implementada para indexação primária por ID:

- **Estrutura**: Árvore balanceada com dados apenas nas folhas
- **Ordem configurável**: Definida pelo usuário na inicialização
- **Operações**: Inserção, busca e impressão da estrutura
- **Complexidade**: O(log n) para todas as operações

A classe `Pagina` representa os nós da árvore, armazenando elementos, offsets e ponteiros para filhos.

3.2.2 Hash Extensível

Implementado para acesso direto por ID:

- **Diretório**: Array de ponteiros para buckets
- **Profundidade global**: Controla o tamanho do diretório
- **Buckets**: Armazenam os registros com profundidade local
- **Expansão dinâmica**: Duplicação do diretório quando necessário

A função hash utilizada garante distribuição uniforme:

```java
protected int hash(int chave) {
    int hash = chave;
    hash = (hash ^ (hash >>> 16)) & 0x7fffffff;
    return hash % (int) Math.pow(2, profundidadeGlobal);
}
```

3.2.3 Lista Invertida

Implementada para buscas por atributos secundários:

- **Dicionário**: Mapeia termos para endereços de blocos
- **Blocos**: Contêm listas de offsets dos registros
- **Atributos indexados**: Temporada, episódios, estúdio, nota, gêneros
- **Buscas combinadas**: Interseção de múltiplos critérios

3.3 TP3 - Compressão e Casamento de Padrões

3.3.1 Algoritmos de Compressão

Huffman:

- **Princípio**: Codificação baseada na frequência dos caracteres
- **Implementação**: Construção da árvore de Huffman e geração de códigos
- **Vantagem**: Eficiente para dados com distribuição desigual de frequências
- **Escolha**: Ideal para textos onde alguns caracteres são muito mais frequentes

LZW (Lempel-Ziv-Welch):

- **Princípio**: Substituição de sequências repetidas por códigos
- **Dicionário dinâmico**: Construído durante a compressão
- **Vantagem**: Eficiente para dados com padrões repetitivos
- **Escolha**: Ideal para dados estruturados como bancos de dados

3.3.2 Algoritmos de Casamento de Padrões

Boyer-Moore:

- **Estratégia**: Busca da direita para a esquerda com saltos inteligentes

- **Pré-processamento**: Tabela de caracteres ruins
- **Complexidade**: O(nm) no pior caso, O(n/m) no melhor caso
- **Escolha**: Eficiente para padrões longos e alfabetos grandes

KMP (Knuth-Morris-Pratt):

- **Estratégia**: Evita comparações desnecessárias usando informações anteriores
- **Pré-processamento**: Tabela de falhas (failure function)
- **Complexidade**: O(n+m) garantido
- **Escolha**: Complexidade linear garantida, ideal para buscas frequentes

3.4 TP4 - Criptografia e Segurança

3.4.1 Cifra de Vigenère

Implementação de criptografia simétrica:

- **Chave**: String alfabética fornecida pelo usuário
- **Operação**: Deslocamento cíclico baseado na chave
- **Aplicação**: Byte a byte em todo o arquivo
- **Escolha**: Simplicidade de implementação e velocidade

```java
public byte[] criptografar(byte[] dados) {
    byte[] resultado = new byte[dados.length];
    int chaveIndex = 0;

    for (int i = 0; i < dados.length; i++) {
        int dadoByte = dados[i] & 0xFF;
        char chaveChar = chave.charAt(chaveIndex % chave.length());
        int chaveValor = chaveChar - 'A';
        int criptografado = (dadoByte + chaveValor) % 256;
        resultado[i] = (byte) criptografado;
        chaveIndex++;
    }
    return resultado;
}
```

3.4.2 RSA

Implementação de criptografia assimétrica:

- **Geração de chaves**: Números primos de 512 bits
- **Chaves numéricas**: Uso exclusivo de BigInteger
- **Processamento em blocos**: Devido às limitações do RSA
- **Escolha**: Segurança robusta com chaves públicas/privadas

Problema Crítico Resolvido: Um bug significativo foi identificado onde `BigInteger.toByteArray()` remove zeros à esquerda, corrompendo dados como o cabeçalho do arquivo. A solução implementada preserva o tamanho original:

```
// Durante criptografia: salva tamanho original
baos.write(bloco.length); // Tamanho original
baos.write((blocosCriptografados.length >> 8) & 0xFF);
baos.write(blocosCriptografados.length & 0xFF);
baos.write(blocosCriptografados);

// Durante descriptografia: restaura tamanho
if (blocosDescriptografados.length < tamanhoOriginal) {
    byte[] blocosCompletos = new byte[tamanhoOriginal];
    System.arraycopy(blocosDescriptografados, 0,
                     blocosCompletos, tamanhoOriginal -
                        blocosDescriptografados.length,
                     blocosDescriptografados.length);
    baos.write(blocosCompletos);
}
```

3.4.3 Integração com CRUD

O sistema de criptografia foi integrado ao CRUD através de:

- **Arquivos temporários**: Descriptografia automática para operação
- **Detecção automática**: Identificação do tipo de criptografia pelo nome do arquivo
- **Sincronização**: Garantia de uso da mesma instância RSA
- **Limpeza automática**: Remoção de arquivos temporários órfãos

4 TESTES E RESULTADOS

4.1 Desempenho dos Índices

Operação	Sequencial	Árvore B+	Hash Extensível	Lista Invertida
Busca por ID	O(n)	O(log n)	O(1)	N/A
Busca por atributo	O(n)	N/A	N/A	O(k)
Inserção	O(1)	O(log n)	O(1) amort.	O(k)
Espaço	O(n)	O(n)	O(n)	O(n·m)

Tabela 1: Complexidade das operações por tipo de índice

4.2 Eficiência da Compressão

Testes realizados com o arquivo `animeDataBase.db` contendo dados reais de animes:

Algoritmo	Tamanho Original	Tamanho Comprimido	Taxa de Compressão
Huffman	3.032 bytes	2.026 bytes	33,18%
LZW	3.032 bytes	1.542 bytes	49,17%

Tabela 2: Resultados reais de compressão obtidos nos testes

Análise dos Resultados:

- O algoritmo **LZW** apresentou melhor taxa de compressão (49,17%), sendo mais eficiente para este tipo de dados estruturados
- O algoritmo **Huffman** obteve taxa de 33,18%, ainda assim significativa para redução de espaço
- Ambos os algoritmos demonstraram eficácia na compressão de dados textuais e estruturados
- O LZW é particularmente eficiente em dados com padrões repetitivos, como campos estruturados de animes

Tempos de Execução dos Algoritmos de Compressão:

Algoritmo	Tempo Compressão	Tempo Descompressão	Eficiência
Huffman	97 ms	57 ms	Rápido
LZW	7.841 ms	78 ms	Lento na compressão

Tabela 3: Tempos de execução reais dos algoritmos de compressão

Observações sobre Performance:

- **Huffman**: Mais rápido na compressão (97ms vs 7.841ms), ideal para aplicações em tempo real
- **LZW**: Apesar de mais lento na compressão, oferece melhor taxa de redução
- **Descompressão**: Ambos apresentam tempos similares e aceitáveis (57-78ms)
- **Trade-off**: Huffman para velocidade, LZW para máxima compressão

4.3 Casamento de Padrões

Testes de busca em strings de nomes de animes:

Algoritmo	Padrão Curto	Padrão Médio	Padrão Longo
Boyer-Moore	Excelente	Bom	Regular
KMP	Bom	Bom	Bom
Força Bruta	Regular	Ruim	Ruim

Tabela 4: Desempenho relativo dos algoritmos de casamento

4.4 Segurança e Performance

Algoritmo	Velocidade	Segurança	Complexidade
Vigenère	Muito Rápida	Baixa	Simples
RSA	Lenta	Alta	Complexa

Tabela 5: Comparação dos algoritmos de criptografia

4.5 Justificativas das Escolhas

Estruturas de Dados:

- **Árvore B+**: Escolhida por manter dados ordenados e garantir acesso logarítmico
- **Hash Extensível**: Selecionado para acesso em tempo constante com expansão dinâmica
- **Lista Invertida**: Necessária para indexação de atributos não-chave

Algoritmos de Compressão:

- **Huffman**: Eficiente para dados textuais com distribuição não-uniforme
- **LZW**: Superior para dados estruturados com repetições

Algoritmos de Busca:

- **Boyer-Moore**: Rápido para padrões longos em alfabetos grandes
- **KMP**: Complexidade linear garantida, confiável

Algoritmos de Criptografia:

- **Vigenère**: Demonstração de criptografia simétrica clássica
- **RSA**: Exemplo de criptografia assimétrica moderna

5 CONCLUSÃO

O desenvolvimento deste sistema demonstrou a evolução natural de um projeto de software, partindo de uma implementação básica até um sistema robusto e completo. Cada trabalho prático introduziu conceitos fundamentais que se complementaram para formar uma solução integrada.

Principais conquistas:

- Implementação bem-sucedida de estruturas de dados clássicas
- Melhoria dramática na eficiência através da indexação (de O(n) para O(1) e O(log n))
- Redução efetiva do espaço de armazenamento via compressão (até 49,17% com LZW)
- Garantia de segurança dos dados através de criptografia robusta
- Sistema modular permitindo uso independente de cada funcionalidade

Desafios superados:

- Gerenciamento complexo de múltiplos arquivos binários
- Sincronização entre índices durante operações de modificação
- Integração transparente da criptografia sem impactar performance
- Correção de bugs sutis na implementação RSA (preservação de zeros à esquerda)
- Tratamento de casos especiais em estruturas dinâmicas (duplicação de diretório)

Lições aprendidas:

- A importância da escolha adequada de estruturas de dados para cada tipo de operação
- O impacto transformador da indexação na performance de sistemas de dados
- A necessidade de testes extensivos para identificar bugs em cenários específicos
- A importância da modularidade para facilitar manutenção e extensões futuras
- O valor da documentação clara para compreensão e manutenção do código

Impacto educacional: Este projeto proporcionou experiência prática com:

- Manipulação de arquivos binários em baixo nível
- Implementação de estruturas de dados avançadas
- Algoritmos clássicos de ciência da computação
- Técnicas de otimização e análise de performance

- Integração de sistemas complexos

O sistema final representa uma solução completa para gerenciamento de dados, incorporando técnicas modernas de armazenamento, indexação, compressão e segurança. A experiência obtida no desenvolvimento deste projeto fornece uma base sólida para enfrentar desafios similares em projetos futuros de maior escala e complexidade. Em conclusão, este projeto demonstrou com sucesso a aplicação prática de conceitos fundamentais de estruturas de dados e algoritmos, resultando em um sistema funcional e educativo que serve como excelente base para desenvolvimentos futuros mais complexos.