Documentação Trabalhos Práticos AEDS III

Rodrigo D. Macedo, Tiago L. Antunes

[1]Instituto de Ciências da Computação – Pontifícia Universidade Católica, Minas Gerais (PUCMG)
Belo Horizonte – MG – Brasil

{rodrigodm2810@gmail.com, tiago-antunes@hotmail.com

Abstract. *This report is presented as documentation of the practical works of the course Algorithms and Data Structures III. The works are divided into TP1, TP2, TP3, TP4 and were developed throughout the semester, covering all the material taught during this period by Professor Hayala Curto.*

Resumo. *Este relatório se apresenta como documentação final dos trabalhos práticos da disciplina de Algoritmos e Estruturas de Dados III. Os trabalhos são divididos em TP1, TP2, TP3, TP4 e foram desenvolvidos ao longo do semestre, cobrindo toda a matéria ministrada durante esse período de tempo, pelo professor Hayala Curto.*

1. Introdução

Com o início da disciplina, foi especificado seu enfoque, que é a manipulação de dados e registros em memória secundária, especificamente em arquivos binários com extensão '.db'. Dessa forma, cada trabalho prático se especializa em uma área da manipulação de arquivos, começando com arquivos sequenciais no TP1, passando para arquivos indexados em forma de estruturas de dados em memória secundária no TP2, compactação de arquivos no TP3 e, finalmente, criptografia e casamento de padrões no TP4.

A divisão dos trabalhos práticos foi realizada da seguinte maneira:

- TP1: Manipulação de Arquivos Sequenciais - Criação da base de dados. - Manipulação de registros através de acesso sequencial. - Implementação e documentação em vídeo.

- TP2: Manipulação de Arquivos Indexados - Implementação de estruturas de dados Árvore B+, Hash e Lista Invertida. - Indexação de registros para acesso eficiente. - Implementação e documentação em vídeo.

- TP3: Compactação de Arquivos - Implementação de algoritmos de compactação Huffman e LZW. - Redução do espaço de armazenamento necessário para os dados. - Implementação e documentação em vídeo.

- TP4: Criptografia e Casamento de Padrões - Implementação de técnicas de criptografia para garantir a segurança dos dados. - Desenvolvimento de algoritmos para casamento de padrões. - Implementação e relatório final.

Ao final de cada um dos trabalhos, o grupo foi responsável por gravar um vídeo explicando o funcionamento do código e o raciocínio por trás das escolhas feitas, apresentados ao Professor Hayala. Esses vídeos foram uma parte essencial do processo de avaliação, permitindo uma demonstração prática das soluções implementadas e a justificativa das decisões técnicas.

Neste relatório, cada um desses vídeos será descrito em forma de texto, apresentando o mesmo processo criativo e de construção dos trabalhos. Serão detalhadas as abordagens utilizadas, os desafios encontrados e como foram superados, bem como os resultados obtidos. Além disso, este documento fornecerá uma visão geral das técnicas e algoritmos empregados, destacando como cada trabalho contribuiu para o entendimento e a aplicação prática dos conceitos de manipulação de arquivos em memória secundária.

Este relatório serve como uma documentação abrangente dos trabalhos práticos realizados, proporcionando uma visão detalhada das metodologias adotadas, das implementações desenvolvidas e dos conhecimentos adquiridos ao longo do semestre.

2. Trabalhos Práticos

2.1. Base de dados

Para o desenvolvimento do trabalho prático 1 e posteriores, foi escolhida uma base dados sobre filmes e séries da Netflix, a qual foi retirada da plataforma Kaggle, contendo um total de 8810 registros.

A base citada acima possui um total de 5 atributos para cada registro, sendo eles: id(long), type(String), title(String), director(String) e dateAdded(String).

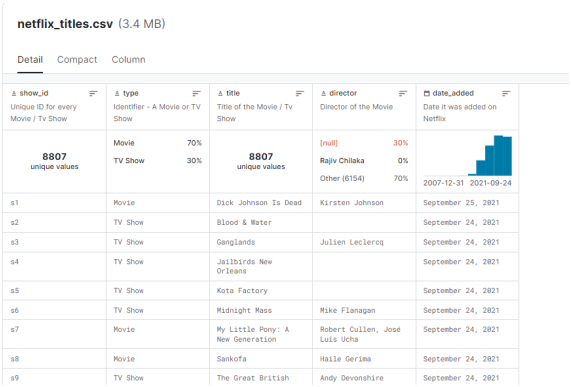

Figure 1. Base de dados

2.2. Trabalho Prático 1

Para o trabalho prático 1, o objetivo foi manipular os registros da base dados, os quais foram armazenados em um arquivo '.db', usando a linguagem Java. O foco foi a criação de um CRUD para manipulação dos registros em um arquivo sequencial. Segue abaixo uma descrição das estruturas de dados, componentes e classes utilizados no projeto.

FileReader e BufferedReader para leitura de um arquivo .csv contendo os dados iniciais a serem armazenados. FileOutputStream e DataOutputStream para escrever os dados lidos do .csv no arquivo binário .db. RandomAccessFile para acessar e manipular registros individuais dentro do arquivo binário.

Os registros foram representados pela classe Netflix, que incluía métodos para ler dados de uma linha do arquivo .csv, converter um objeto para um array de bytes e vice-versa. Cada registro armazenado no arquivo .db incluía um campo de lápide (indicando

se o registro foi deletado), o tamanho do registro e os dados do registro em si. Funcionalidades Implementadas

O código começa lendo o arquivo netflix.csv, ignorando a primeira linha (cabeçalho). Cada linha subsequente é convertida em um objeto Netflix e escrita no arquivo data.db com um campo de lápide inicializado como false, seguido pelo tamanho do registro e os dados do registro. O último ID utilizado é armazenado no início do arquivo para facilitar a geração de novos IDs. Menu de Interação com o Usuário:

Um menu interativo é apresentado ao usuário, permitindo adicionar, ler, atualizar e deletar registros, além de sair do programa.

Create: Coleta dados do usuário, cria um novo objeto Netflix, e o escreve no final do arquivo data.db. O ID do novo registro é incrementado a partir do último ID armazenado. Read: Lê o arquivo data.db de forma sequencial para encontrar e exibir um registro com um ID específico. Ignora registros marcados como deletados. Update: Atualiza um registro existente. Se o novo registro for maior que o antigo, o antigo é marcado como deletado, e o novo é adicionado ao final do arquivo. Delete: Marca um registro específico como deletado.

2.3. Trabalho Prático 2

Para o trabalho prático 2, foram implementadas estruturas de dados complexas para otimizar o acesso e manipulação de registros em memória secundária. As estruturas escolhidas foram Árvore B+, Hash Extensível e Lista Invertida. Segue abaixo uma descrição das estruturas de dados, componentes e classes utilizados no projeto.

Hashing Extensível (Hash)

A estrutura de dados Hashing Extensível foi utilizada para gerenciar e distribuir registros em buckets, permitindo uma expansão dinâmica da tabela de hash para evitar colisões.

Componentes e Funcionalidades:

Construtor: Inicializa os arquivos de dados e diretório, carregando dados existentes ou criando novos. Se persistir for verdadeiro, os valores de pGlobal, tamDir e o diretório são lidos dos arquivos existentes. Caso contrário, inicializa esses valores.

Métodos Principais: insertHash(int id, long end): Insere um novo registro na tabela de hash. Duplica o diretório se necessário e reorganiza os buckets para evitar colisões. searchHash(int id): Busca um registro pelo ID, retornando a posição do registro no arquivo de dados. updateHash(int id, long end): Atualiza o ponteiro de um registro existente. deleteHash(int id): Remove um registro, marcando-o como deletado.

Exemplo de Funcionamento:

Inserção: Quando um novo registro é inserido, a função insertHash calcula o índice hash e tenta inserir o registro no bucket correspondente. Se o bucket estiver cheio, a tabela de hash é expandida e os registros são redistribuídos. Busca: A função searchHash usa o índice hash para localizar o bucket apropriado e percorre o bucket para encontrar o registro desejado.

Árvores B+ (ArvoreBP)

A estrutura de dados Árvore B+ foi utilizada para indexação eficiente de registros, permitindo buscas, inserções e atualizações rápidas em grandes volumes de dados.

Componentes e Funcionalidades:

Construtor:

Inicializa a árvore a partir de um arquivo ou cria uma nova estrutura. Se o arquivo estiver vazio, uma nova árvore é criada com a raiz definida como nula. Caso contrário, a raiz é lida do arquivo. Métodos Principais:

inserir(int id, long ptrDados): Insere um novo registro na árvore. Se o nó está cheio, realiza o split e promove chaves ao nó pai. buscar(int id): Busca um registro pelo ID, retornando o ponteiro para os dados. atualizar(int id, long ptrDados): Atualiza o ponteiro de um registro existente. Exemplo de Funcionamento:

Inserção: Quando um novo registro é inserido, a função inserir localiza a folha apropriada para a inserção. Se a folha estiver cheia, a função split é chamada para dividir o nó e promover uma chave ao nó pai. Busca: A função buscar percorre a árvore a partir da raiz, descendo pelos nós até encontrar o registro com o ID desejado.

Lista Invertida (ListaInvertida)

A estrutura de dados Lista Invertida foi utilizada para associar termos a IDs de registros, facilitando buscas rápidas por palavras-chave.

Componentes e Funcionalidades:

Construtor:

Inicializa os arquivos de índice e dados. Se os arquivos não existirem, cria novos. Métodos Principais:

criaLista(ArrayList¡String¿ termos, int idFilme): Cria uma lista de termos associada a um ID de filme. Adiciona novos termos ao índice e insere IDs na lista encadeada. adicionaTermo(String termo): Adiciona um novo termo ao arquivo de índice. insereId(String termo, int id): Insere um ID na lista encadeada associada a um termo existente. pegaIds(String termo): Retorna todos os IDs associados a um termo. Exemplo de Funcionamento:

Inserção: A função criaLista percorre os termos fornecidos e chama insereId para associar cada termo ao ID do filme. Se o termo não existir, adicionaTermo é chamada para adicionar o termo ao índice. Busca: A função pegaIds busca o termo no índice e recupera todos os IDs associados a ele através da lista encadeada.

2.4. Trabalho Prático 3

O trabalho prático 3 teve como enfoque a compactação de arquivos e a implementação de técnicas para esse processo. O objetivo da compressão é otimizar o armazenamento de registros em arquivos binários, como forma de simular processos que ocorrem na vida real. As técnincas implementadas foram os algortimos LZW e Huffman. Segue abaixo uma descrição das estruturas de dados, componentes e classes utilizados no projeto.

Algoritmo de Compressão de Huffman

O algoritmo de Huffman é uma técnica de compressão de dados sem perdas que

utiliza a frequência de ocorrência dos caracteres para criar um código binário prefixo, mais curto para os caracteres mais frequentes.

Componentes e Funcionalidades:

Métodos Principais:

compactacao(String compactacaoPath, int num): Realiza a compressão dos dados utilizando o algoritmo de Huffman, armazena a árvore de Huffman utilizada e calcula a taxa de compressão. descompactacao(String compressedPath, int num): Descomprime os dados utilizando a árvore de Huffman armazenada. gerarCodigos(No no, String codigo, HashMap¡Byte, String¿ codigos): Gera os códigos binários para cada byte com base na árvore de Huffman. armazenaArvore(No no, String arvorePath): Armazena a árvore de Huffman em um arquivo. recuperarArvore(No no, long pos, String arvorePath): Recupera a árvore de Huffman de um arquivo. calculaTaxa(int tamOriginal, int tamComprimido): Calcula a taxa de compressão.

Exemplo de Funcionamento:

Compressão: O método compactacao lê o arquivo de dados, calcula a frequência de cada byte, gera a árvore de Huffman, cria os códigos binários, e escreve os dados comprimidos em um novo arquivo. Descompressão: O método descompactacao lê o arquivo comprimido, utiliza a árvore de Huffman armazenada para decodificar os bytes e escreve os dados descomprimidos de volta no arquivo original.

Algoritmo de Compressão LZW

O algoritmo de compressão LZW (Lempel-Ziv-Welch) é uma técnica de compressão sem perdas que substitui sequências de caracteres por códigos de comprimento fixo, utilizando um dicionário de sequências já encontradas.

Componentes e Funcionalidades:

Constantes: BITSPORINDICE: Define a quantidade de bits por índice no dicionário.

Métodos Principais: compactacao(String camArqSaida): Realiza a compressão dos dados utilizando o algoritmo LZW, cria um dicionário de sequências e armazena os índices comprimidos. descompactacao(String camCompactado): Descomprime os dados utilizando o dicionário de sequências armazenado. calculaTaxa(int tamOriginal, int tamComprimido): Calcula a taxa de compressão.

Exemplo de Funcionamento:

Compressão: O método compactacao lê o arquivo de dados, inicializa um dicionário com todos os bytes possíveis, processa o texto para encontrar sequências no dicionário, substitui essas sequências por índices, e escreve os dados comprimidos em um novo arquivo. Descompressão: O método descompactacao lê o arquivo comprimido, reconstrói o dicionário de sequências a partir dos índices, decodifica os índices de volta para as sequências originais, e escreve os dados descomprimidos no arquivo original.

Classe BitSequence

A classe BitSequence gerencia um vetor de bits para armazenar sequências de números em forma compacta.

Atributos: bitsPorNumero: Quantidade de bits por número. ultimoBit: Próximo bit a ser usado. bs: Vetor de bits.

Métodos: add(int n): Adiciona um número de ”bitsPorNumero” bits ao vetor de bits. get(int i): Recupera um número de ”bitsPorNumero” bits na posição i. size(): Retorna a quantidade de números armazenados no BitSet. getBytes(): Retorna os bytes armazenados no BitSet. setBytes(int n, byte[] bytes): Define os bytes no BitSet.

Classe CsvReader

A classe CsvReader é responsável por ler arquivos CSV e retornar seu conteúdo como uma string.

Método: readCsvToString(String filePath): Lê um arquivo CSV e retorna seu conteúdo como uma string.

Classe Principal TP

A classe principal TP gerencia o menu interativo para o usuário realizar operações de CRUD (Create, Read, Update, Delete) e compressão/descompressão de dados.

Métodos Principais: start(): Inicializa o programa, permitindo ao usuário escolher entre continuar com a base de dados atual ou iniciar uma nova. menu(): Apresenta o menu de opções para o usuário. create(Netflix netflix): Adiciona um novo registro na base de dados. read(int id): Lê um registro da base de dados. update(Netflix netflix): Atualiza um registro na base de dados. delete(int id): Deleta um registro da base de dados.

2.5. Trabalho Prático 4

O trabalho prático 4 teve como objetivo implementar técnicas de criptografia e casamento de padrões para aumentar a segurança e eficiência na busca de dados. As técnicas utilizadas foram o algoritmo RSA para criptografia e o algoritmo de Boyer-Moore para casamento de padrões. A seguir, um resumo das funcionalidades e componentes utilizados no código.

Criptografia RSA

O RSA é um algoritmo de criptografia assimétrica amplamente utilizado para garantir a segurança de dados. Ele utiliza um par de chaves (pública e privada) para criptografar e descriptografar mensagens.

Componentes e Funcionalidades:

Classe Rsa:

Atributos:

p, q: Números primos utilizados na geração das chaves. n, e, d: Componentes das chaves pública e privada. Métodos Principais: generateKeys(int bitLen): Gera as chaves pública e privada com o tamanho especificado em bits. encrypt(BigInteger message): Criptografa uma mensagem utilizando a chave pública. decrypt(BigInteger encryptedMessage): Descriptografa uma mensagem utilizando a chave privada. generateRandomPrime(int bitLen): Gera um número primo aleatório com o tamanho especificado em bits. mdc(BigInteger a, BigInteger b): Calcula o máximo divisor comum entre dois números. achaE(BigInteger z): Encontra um valor para e que seja coprimo de z.

Exemplo de Funcionamento:

Criptografia: O método encrypt eleva a mensagem à potência e e tira o módulo n, resultando na mensagem criptografada. Descriptografia: O método decrypt eleva a mensagem criptografada à potência d e tira o módulo n, resultando na mensagem original. Algoritmo de Casamento de Padrões Boyer-Moore O algoritmo de Boyer-Moore é uma técnica eficiente para encontrar substrings dentro de uma string maior. Ele utiliza duas heurísticas principais: a regra do caractere ruim e a regra do sufixo bom.

Casamento de Padrões Boyer-Moore

Métodos Principais: pesquisaCaractereRuim(char[] txt, char[] pat): Realiza a busca utilizando a regra do caractere ruim. pesquisaSufixoBom(char[] text, char[] pat): Realiza a busca utilizando a regra do sufixo bom. caractereRuim(char[] str, int size, int[] caractereRuim): Preenche o vetor de ocorrência do caractere ruim. sufixoBom(int[] shift, int[] bpos, char[] pat, int m): Calcula o próximo sufixo bom do padrão e atualiza o vetor bpos. sufixoBom2(int[] shift, int[] bpos, char[] pat, int m): Implementa o caso 2 da regra de sufixo bom.

Exemplo de Funcionamento:

Caractere Ruim: A função pesquisaCaractereRuim move o padrão ao longo do texto para pular as seções onde o caractere do texto não corresponde ao do padrão. Sufixo Bom: A função pesquisaSufixoBom move o padrão ao longo do texto utilizando informações sobre os sufixos do padrão que já foram correspondidos.

Classe Principal TP

A classe principal TP gerencia o menu interativo para o usuário realizar operações de CRUD (Create, Read, Update, Delete), compressão/descompressão de dados, criptografia/descriptografia e casamento de padrões.

Métodos Principais:

start(): Inicializa o programa, permitindo ao usuário escolher entre continuar com a base de dados atual ou iniciar uma nova. menu(): Apresenta o menu de opções para o usuário. create(Netflix netflix): Adiciona um novo registro na base de dados. read(int id): Lê um registro da base de dados. update(Netflix netflix): Atualiza um registro na base de dados. delete(int id): Deleta um registro da base de dados. menu(): Apresenta o menu de opções ao usuário, incluindo opções de compactação, descompactação, criptografia, descriptografia e casamento de padrões. Exemplo de Funcionamento:

CRUD: As funções create, read, update e delete permitem ao usuário manipular os registros na base de dados. Compactação/Descompactação: As funções de compactação e descompactação permitem ao usuário reduzir o tamanho da base de dados utilizando os algoritmos de Huffman e LZW.

Criptografia/Descriptografia: As funções de criptografia e descriptografia utilizam o algoritmo RSA para proteger os dados na base de dados.

Casamento de Padrões: A função de casamento de padrões utiliza o algoritmo de Boyer-Moore para encontrar substrings dentro dos registros na base de dados.

I

3. Testes e Resultados

Para a execução dos testes dos algortimos e programas criados ao longos dos projetos, foram utilizados os sistemas operacionais Windows e Linux, com ambos fazendo uso da IDE VIsual Studio Code para construir e testar os programas.

Inicialmente, o CRUD criado no TP1 se mostra extremamente eficiente dentro de sua limitação de acesso sequencial, realizando todas as operações corretamente e sendo capaz de manter as restrições e padrões iniciais propostos.

No TP2, a implementação das estruturas Hash Extensível foi um sucesso para no quesito manipulação de dados, se mostrando porém, lenta ao criar os arquivos indexados, uma vez que necessitam iterar por toda a base dados. Porém, a estrutura da Árvore B+ encontrou problemas em relação a deleção e atualização de dados, uma vez que a o processo de reorganização de nós, após a deleção de um registro, se mostrou ineficiente ao cumprir sua tarefa.

No TP3, os algoritmos Huffman e LZW se apresentaram extremamente eficientes ao compactar os arquivos, com ambos atingindo uma taxa de compressão de aproximadamente 55. Além disso, mesmo com a necessidade de acessar sequencialmente todos os registros do arquivo, os algoritmos se mostraram mais rápidos se comparados a criação das estruturas de dados do TP2.

No TP4, o algoritmos Boyer-Moore se mostrou extremamente eficente na busca por padrões no texto, retornando todas as posições do padrão no texto rapidamente. Já o algoritmo RSA, pela necessidade de realizar todas as contas matemáticas para a criptografia de cada caractere acabou por ter um desempenho inferior ao Boyer-Moore, desempenho esse completamente dentro do esperado, já que o primeiro somente percorre os caracteres, enquanto o outro realiza diversas operações a cada iteração. Além disso, ao utilizar os valores dentro do padrão para o RSA, com números que tornem as chaves inquebráveis, se percebe que o arquivo de saída é pelo menos 10 vezes maior que o arquivo original.

4. Conclusão

Ao longo deste relatório, foram detalhados os algoritmos e estruturas de dados utilizados nos trabalhos práticos da disciplina de Algoritmos e Estruturas de Dados III. O enfoque principal da disciplina foi a manipulação de arquivos em memória secundária, o que permitiu a aplicação prática de conceitos teóricos em diferentes contextos, como manipulação de arquivos sequenciais, uso de estruturas de dados complexas como árvores B+, listas invertidas e hashing, além de técnicas de compressão e criptografia de dados.

Os testes realizados em cada módulo demonstraram a correção, desempenho e eficiência das implementações. Os resultados obtidos mostraram que: - Os algoritmos de compressão foram eficazes na redução do tamanho dos arquivos, com diferentes taxas de compressão dependendo do algoritmo e do tamanho do arquivo original. - As estruturas de dados utilizadas no TP2 proporcionaram buscas rápidas e eficientes, mesmo com grandes volumes de dados, porém a construção dos arquivos de índices se mostrou mais lenta que o esperado. - A criptografia RSA implementada no TP4 garantiu a criptografia dos dados, tornando a descriptografia possível com uso das chaves, enquanto o algoritmo Boyer-Moore permitiu buscas rápidas e precisas de padrões no arquivo.

Os testes de desempenho revelaram as forças e limitações de cada abordagem. Por exemplo, o algoritmo Huffman mostrou-se mais eficiente em termos de tempo de execução para arquivos menores, enquanto o LZW foi mais eficaz para arquivos maiores.

A realização destes trabalhos práticos consolidou o entendimento dos conceitos teóricos da disciplina, proporcionando uma experiência prática na manipulação e otimização de dados em memória secundária.