Relatório Trabalhos Práticos Aeds3

Andrei Massaini, Luiz Fernando

December 16, 2022

1 Trabalhos Práticos Algoritmos e estrutura de dados 3

1.1 Resumo:

Este artigo descreve os resultados dos trabalhos práticos feitos na disciplina de Algoritmos e estruturas de dados III, do curso de Ciência da Computação da Puc Minas.

1.2 Introdução:

Os trabalhos práticos da disciplina estavam inseridos em um contexto de conta bancária. Como especificação do projeto, deveríamos respeitar o limite de memória primária, e realizar operações no arquivo em memória secundária. Após isso, deveríamos aplicar os algoritmos estudados nesse mesmo contexto, como o de criptografia, casamento de padrões e a ordenação do arquivo.

2 Desenvolvimento, testes e resultados

2.1 Trabalhando com bytes:

Uma das especificações do projeto, foi de escrever as contas bancárias no arquivo utilizando-se por meio da serialização do objeto em bytes. Para isso, foi necessário converter as instancias das contas em vetores de bytes, e por fim, escreve-las no arquivo. Para isso, dois métodos essenciais foram implementados: ConverteContaEmByte() e o DecodificaContaEmBteArray().

```java
public byte[] converteContaEmByte()throws IOException{
    //Este metodo converte a conta instanciada em um vetor de byte.
    ByteArrayOutputStream vetorByte = new ByteArrayOutputStream();//Escrita
    DataOutputStream buffer = new DataOutputStream(vetorByte);//Escrita|buffer
    buffer.writeInt(idConta);
    buffer.writeUTF(nomePessoa);
    buffer.writeUTF(email);
    buffer.writeUTF(nomeUsuario);
    buffer.writeUTF(senha);
    buffer.writeUTF(cpf);
    buffer.writeUTF(cidade);
    buffer.writeInt(transferenciasRealizadas);
    buffer.writeFloat(saldoConta);
    return vetorByte.toByteArray();
}
```

Método que retorna uma conta em um vetor de byte.

```java
public void decodificaByteArray(byte[] vetorByte)throws IOException{
    //Este metodo recebe um vetor de byte do arquivo e converte o mesmo em um objeto do tipo Conta.
    ByteArrayInputStream bufferParaLeitura = new ByteArrayInputStream(vetorByte);
    DataInputStream leitura = new DataInputStream(bufferParaLeitura);
    idConta = leitura.readInt();
    nomePessoa = leitura.readUTF();
    email = leitura.readUTF();
    nomeUsuario = leitura.readUTF();
    senha = leitura.readUTF();
    cpf = leitura.readUTF();
    cidade = leitura.readUTF();
    transferenciasRealizadas = leitura.readInt();
    saldoConta = leitura.readFloat();

}
```

Método que recebe um vetor de byte e 'converte' ele em um objeto do tipo conta.

2.2 Escrevendo no arquivo:

O acesso ao arquivo, feito de forma aleatório utilizando-se da classe 'RandomAcessFile' do java. Para escrever a conta no arquivo, foi necessário seguir os princípios apresentados na disciplina, como a adição de um indicador do ultimo ID da conta inserida no cabeçalho do arquivo,indicador de bytes a serem lidos por cada conta e o uso de lápides para realizar a remoção lógica das contas.

```java
public static void writeAccount(Conta conta) {
    try {
        RandomAccessFile arquivo = new RandomAccessFile(nomeArquivo, mode: "rw");
        byte[] array;
        array = conta.converteContaEmByte();
        arquivo.seek(pos: 0);
        arquivo.writeInt(conta.idConta);
        arquivo.seek(arquivo.length());
        arquivo.writeChar(v: ' ');
        arquivo.writeInt(array.length);
        arquivo.write(array);
        arquivo.close();
    } catch (Exception e) {
        System.out.println(e);
        System.out.println(x: "Erro ao criar conta!");
    }
}
```

Método que escreve uma conta no arquivo.

3 Algoritmos implementados:

3.1 Intercalação balanceada:

O algoritmo de ordenação escolhido foi o de intercalação balanceada. Onde teríamos que respeitar o limite da memória primária para ordenar o conjunto de dados. Para isso, primeiramente foi necessário realizar a distribuição dos arquivos em N caminhos parametrizados. Após isso, realizar a intercalação trazendo apenas N contas para a memória principal, ordenando-as em cadeia.

```java
public static int distribuicao(int ram, int caminhos) {
    /*Realiza a distribuição de acordo com o tamanho suportado pela ram (tam) e a quantidade de caminhos especific
     * por parametro.
     */
    int quantidade = 0;
    try {
        List<Conta> contas = new ArrayList<>();
        RandomAccessFile[] temp = new RandomAccessFile[caminhos];
        for (int i = 0; i < caminhos; i++) {
            temp[i] = new RandomAccessFile("output/tmp" + i + PREFIXO, mode: "rw");
        }
        while (ptrControl != -1) {
            for (int i = 0; i < caminhos; i++) {
                for (int j = 0; j < ram; j++) {
                    var conta = readFile(nomeArquivo);
                      System.out.println(conta);
                    if (conta != null) {
                        contas.add(conta);
                    }
                }
                Collections.sort(contas);
                for (Conta conta : contas) {
                    byte[] ba = conta.converteContaEmByte();
                    temp[i].writeChar(v: ' ');
                    temp[i].writeInt(ba.length);
                    temp[i].write(ba);
                }
                quantidade += contas.size();
                contas.clear();
            }
        }
        for (var t : temp) {
            t.close();
        }
    } catch (Exception e) {
        System.out.println("Erro dist. " + e.getMessage());
        e.printStackTrace();
    }
    return quantidade;//Retorna a quantidade total de registros do arquivo original
```

Método que realiza a distribuição do arquivo.

```java
public static void intercalacao(int ram, int caminhos, boolean isBase) {
    CustomFile[] temp1 = new CustomFile[caminhos];
    for (int i = 0; i < caminhos; i++) {
        temp1[i] = new CustomFile("output/tmp" + (isBase ? i : i + caminhos) + PREFIXO);
        //Abrindo arquivos originais frutos da distribuição.
}
    CustomFile[] temp2 = new CustomFile[caminhos];
    for (int i = 0; i < caminhos; i++) {
        temp2[i] = new CustomFile("output/tmp" + (isBase ? i + caminhos : i) + PREFIXO);
        //A brindo arquivos temporarios auxiliares para intercalar
    }
    Map<CustomFile, Conta> map = new HashMap<>();
    int tempPos = 0;
    try {
        while (true) {
            for (int i = 0; i < caminhos; i++) {
                if (map.get(temp1[i]) == null && temp1[i].readRegisterSize < ram) {//Verifico se ainda existem registros pra ler
                    Conta conta = temp1[i].readNext();//Instancio a conta lendo do arquivo temp1[i];
                    if (conta != null) {
                        map.put(temp1[i], conta);//Inserindo no hash map.
                    }
                }
            }
            if (map.isEmpty()) break;
            var ordered = map.entrySet().stream().sorted(Map.Entry.comparingByValue()).toList();
            var firstConta = ordered.get(index: 0);
            temp2[tempPos].writeConta(firstConta.getValue());
            if (temp2[tempPos].size == limite) {
                nomeArquivoFinal = temp2[tempPos].fileName;
                break;
            }
            if (temp2[tempPos].size % (ram * caminhos) == 0) {
                tempPos++;
                if (tempPos == caminhos) {
                    tempPos = 0;
                }
                for (var t : temp1) { // limpa a quantidade de registros lidos
                    t.readRegisterSize = 0;
                }
            }
            map.remove(firstConta.getKey());
        }
        for (var t : temp1) t.file.close();
        for (var t : temp2) t.file.close();
    } catch (Exception e) {
```

Método que realiza a intercalação do arquivo.

3.2 Arvore B+:

Também foi implementado o armazenamento das contas em uma árvore B+, simulando a lógica de um banco de dados. No método de inserção, inserimos a posição da conta no arquivo que atuaria como um índice, e como a busca nesse tipo de árvore é otimizada o tempo de busca era extremamente reduzido como consequência.

```java
public void criaArvore() {
  try {
    // Loopa pelo arquivo de dados inserindo sua chave/valor na árvore:
    RandomAccessFile raf = new RandomAccessFile(name: "output/conta.db", mode: "rw");
    raf.seek(pos: 4);
    long pos;
    char lapide;
    int tamanho;
    byte[] ba;
    while (raf.getFilePointer() != -1) {
      pos = raf.getFilePointer();
      lapide = raf.readChar();
      tamanho = raf.readInt();
      ba = new byte[tamanho];
      raf.read(ba);
      if (lapide != '*') {
        Conta conta = new Conta();
        conta.decodificaByteArray(ba);
        System.out.println(
            "Inserindo conta de Id: " + conta.idConta + ", e sua posicao no arquivo eh: " + pos);
        this.insert(conta.idConta, pos);
      }
    }
    raf.close();
  } catch (Exception e) {
  }
```

Criação da árvore inserindo os ponteiros do arquivos para as contas criadas.

3.3 Compressão de textos: Algoritmo de Huffman

O algoritmo de Huffman foi implementado no projeto. Nesse algoritmo, realizamos a codificação do texto contido no conjunto de arquivos, assim como sua decodificação. Para a codificação, primeiramente percorríamos o arquivo em bytes e extraímos seu texto. Na decodificação, utilizamos o arquivo de códigos gerados pela codificação e utilizamos o mesmo para transcrever a mensagem e escreve-la novamente em byte em um novo arquivo.

```java
public static void contentDecoded(No raiz, StringBuilder sb, StringBuilder sb2){
    System.out.print(s: "A string descodificada eh: ");
     if (eFolha(raiz)) {
        // caso especial para entradas como a, aa, aaa, etc.
        while (raiz.frequencia-- > 0) {
            System.out.print(raiz.a);
        }
    } else {
        // Novamente na arvore so que dessa vez descodificar a string codificada
        int index = -1;
        while (index < sb.length() - 1) {
            index = descodificarDado(raiz, index, sb2);
        }
    }
}

public static void codificarDado(No raiz, String str, Map<Character, String> huffmanCode) {
    if (raiz == null) {
        return;
    }
    // verificar se o no eh uma folha
    if (eFolha(raiz)) {
        huffmanCode.put(raiz.a, str.length() > 0 ? str : "1");
    }
    codificarDado(raiz.esq, str + '0', huffmanCode);
    codificarDado(raiz.dir, str + '1', huffmanCode);

}
```

Métodos recursivos de codificação e decodificação de Huffman.

3.4 Casamento de padrão KMP:

Foi implementado no projeto o algoritmo de casamento de padrão KMP, onde o mesmo tem o objetivo de realizar a busca de padrões parametrizados dentro do conjunto de dados 'conta.db'.

```java
class KMP {
    void KMPSearch(String padrao, String texto) {
        int M = padrao.length();
        int N = texto.length();

        // lps[] vai guardar o maior sufixo encontrado
        int lps[] = new int[M];
        int j = 0; // index for padrao[]

        armazenaSufixo(padrao, M, lps);

        int i = 0; // index for texto[]
        while ((N - i) >= (M - j)) {
            if (padrao.charAt(j) == texto.charAt(i)) {
                j++;
                i++;
            }
            if (j == M) {
                System.out.println("Padrao encontrado " + "no indice " + (i - j));
                j = lps[j - 1];
            }

            // mismatch after j matches
            else if (i < N && padrao.charAt(j) != texto.charAt(i)) {
                if (j != 0)
                    j = lps[j - 1];
                else
                    i = i + 1;
            }
        }
    }
```

3.5 Criptografia, ciframento de César:

O método de criptografia utilizado no projeto, foi o ciframento de césar. Ao criar cada conta, a senha era criptografada utilizando-se de uma chave K que seria somada a cada caractere da senha. Ao listar as contas, a senha era descriptografada no processo.

```java
public static String criptografaSenha(String str) {
    int aux = 0;
    String aux2 = "";
    for (int i = 0; i < str.length(); i++) {
        aux = (int) str.charAt(i);// converte cada char da string em int
        aux += cesarKey; // Aumenta 3 do char convertido em inteiro
        char b = (char) aux;// converte o int incrementado em char.
        aux2 += String.valueOf(b);// armazena o resultado na variavel aux2
    }
    return aux2;
}
public static String descriptografaSenha(String str){
    int aux = 0;
    String aux2 = "";
    for (int i = 0; i < str.length(); i++) {
        aux = (int) str.charAt(i);// converte cada char da string em int
        aux -= cesarKey; // Aumenta 3 do char convertido em inteiro
        char b = (char) aux;// converte o int incrementado em char.
        aux2 += String.valueOf(b);// armazena o resultado na variavel aux2
    }
    return aux2;
}
```

Métodos simples de ciframento e deciframento da senha.

4 Conclusão:

Após a conclusão do projeto, o grupo foi capaz compreender e trabalhar com arquivos no formato binário, e em memória secundária. Além disso,extraímos o conhecimento de diversas técnicas de algoritmos que são banalmente utilizados diariamente em diversos sistemas, porém não tínhamos esta compreensão previamente.

Link do projeto no GitHub: Link