

# Trabalho AEDS III - CRUD Clubes de Futebol

Davidson Marra<sup>1</sup>, Pedro Henrique Teixeira<sup>2</sup>

<sup>1</sup>PUC Minas – Pontifícia Universidade Católica de Minas Gerais (PUCMG)  
30.535-901 – Belo Horizonte – MG – Brazil

**Abstract.** *This work aims to show the application of an algorithm that controls data through secondary memory (text file). As an application model a CRUD scheme was used for football clubs, whereby they can add points and play matches against each other.*

**Resumo.** *Esse trabalho têm como objetivo mostrar a aplicação de um algoritmo que controla os dados por meio da memória secundária (arquivo de texto). Como modelo de aplicação foi utilizado um esquema de CRUD para clubes de futebol, pelo qual eles podem somar pontos e disputar partidas entre si.*

## 1. Introdução

Com o desenvolvimento dessa aplicação, foi possível ver como é na prática o funcionamento de um algoritmo que trabalhe com arquivos de texto. Para a elaboração do CRUD dos clubes e leitura dos arquivos, foi utilizado a linguagem Java.

## 2. Desenvolvimento

Para o desenvolvimento da aplicação, foi separado em arquivos para melhor organização e aproveitamento do código. Esses arquivos foram: CRUD, Clube, Interface Entidade e Menu.

### 2.1. Clube

O arquivo Clube, representa a classe que será a entidade/registo do projeto. Nela estão as propriedades de um Clube, pelo qual são: Byte id, pois por se tratar de uma simulação de um campeonato brasileiro o número de times não pode passar de 20. String nome. String cnpj. String cidade. Byte partidasJogadas, pois o número de partidas não pode passar de 38. Byte pontos, pois um time não pode passar de 114 pontos. Além das propriedades, ele contém 4 métodos. O construtor, pelo qual inicializa as propriedades do objeto. O método toString, onde printa as informações de um clube. O procedimento toByteArray, pelo qual retorna um array de bytes preenchidos para escrever no arquivo. E o método fromByteArray, pelo qual lê o arquivo e salva as informações em um array de Bytes.

### 2.2. CRUD

O arquivo CRUD, armazena todos os métodos que vão gerenciar as operações feitas no arquivo. Elas são: Create, Read, Delete e Update.

### 2.3. Interface Entidade

A interface Entidade tem como objetivo determinar os métodos que obrigatoriamente devem ser implementados na classe Clube.

## 2.4. Menu

A classe Menu, foi criada para desenvolver a interface gráfica do aplicativo e fazer as chamadas dos métodos do CRUD. Além de armazenar o método para jogar uma partida, onde é informado o id dos dois times que iram se enfrentar e o placar da partida. Após isso é atualizado o número de pontos e a quantidade de partidas jogadas dos times.

## 3. Testes e Resultados

Os testes foram feitos em cada procedimento do CRUD, pelo qual é verificado se está certo conforme o resultado final do arquivo de texto que armazena os dados.

### 3.1. Método Criar

Para utilizar o método criar é necessário ter a pasta "dados" criada na raiz do projeto. Ao selecionar a opção 1 no menu, o próximo passo é preencher as informações do clube, da seguinte maneira:

```
Opcao:
1
Você entrou no método criar.
Entre com o nome do Clube:
Atlético
Entre com o cnpj do Clube (sem máscara):
1234567
Entre com o cidade do Clube (sem máscara):
Belo Horizonte
TIME CRIADO COM SUCESSO!
```

Figure 1. Time criado com sucesso

Após o time ser criado de maneira correta, o arquivo "Clube.db" será possível de se acessar dentro da pasta "dados", e nele estão armazenados as informações dos times da seguinte maneira:

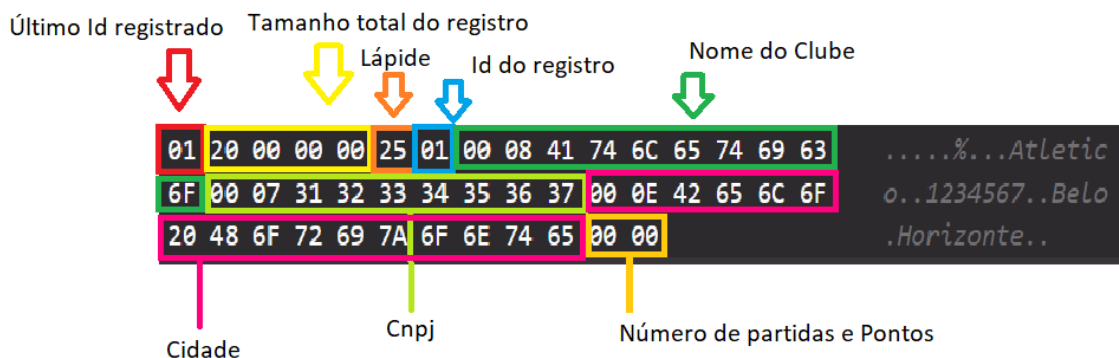


Figure 2. Organização arquivo de dados

### 3.2. Método Listar

Para fazer a listagem das informações de um clube, é necessário passar a opção 2 no menu. E depois informar o id do clube que deseja ser achado, o retorno deve ser:

```

2
Você entrou no método pesquisar.
Entre com o id do Clube:
1
TIME ENCONTRADO:

Nome: Atletico
CNPJ: 1234567
Cidade: Belo Horizonte
Partidas Jogadas: 0
Pontos: 0

```

Figure 3. Retorno método Listar

### 3.3. Método Atualizar

Para atualizar os dados de um clube, no menu é selecionado a opção 3 e após isso é indicado o id do clube e as suas novas informações. Ao completar as informações, caso tudo dê certo é printado os novos dados do clube, da seguinte forma:

```

Opcao:
3
Você entrou no método atualizar.
Entre com o id do Clube:
1
Entre com o nome do Clube:
Vila Nova
Entre com o cnpj do Clube (sem máscara):
1234
Entre com o cidade do Clube (sem máscara):
Nova Lima

Nome: Vila Nova
CNPJ: 1234
Cidade: Nova Lima
Partidas Jogadas: 0
Pontos: 0
TIME ALTERADO

```

Figure 4. Método Atualizar

Caso as informações do novo clube sejam maiores que a do clube antigo, o arquivo primeiro vai marcar o antigo como excluído e após isso escrever as informações do novo com o mesmo id, da seguinte forma:

```

01 2a 00 00 00 11 01 00 04 67 61 6c 6f 00 03 31 .*.....galo..1
32 33 00 02 42 48 00 00 20 00 00 00 2b 01 00 0d 23..BH.. ...+...
56 61 73 63 6f 20 64 61 20 47 61 6d 61 00 08 31 Vasco da Gama..1
32 33 34 35 36 37 38 00 0e 52 69 6f 20 64 65 20 2345678..Rio de
4a 61 6e 65 69 72 6f 00 00 Janeiro..

```

Figure 5. Método Atualizar com infos maiores

### 3.4. Método Excluir

O método excluir pode ser utilizado selecionando a opção 4 no menu e passando o id do clube que deseja ser excluído, dessa forma:

```
Opcao:
4
Você entrou no método Exclui.
Entre com o id do Clube:
1
TIME EXCLUIDO:

Nome: Vila Nova
CNPJ: 1234
Cidade: Nova Lima
Partidas Jogadas: 0
Pontos: 0
```

Figure 6. Método Excluir

No registro do clube excluído é trocado o tamanho do arquivo por um "\*" para indicar que ele não existe mais, ficando da seguinte forma:

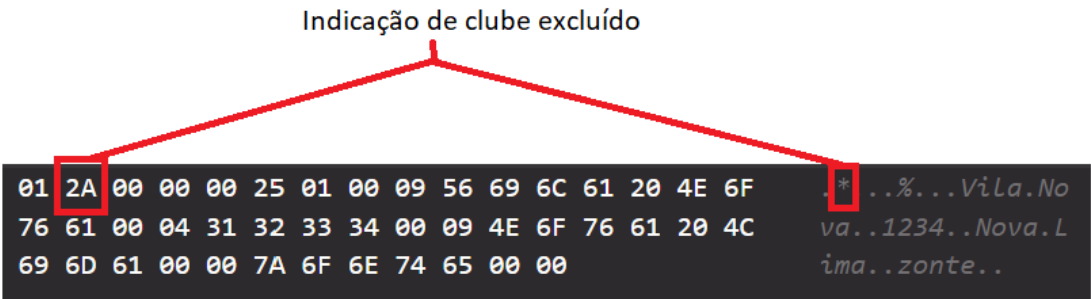


Figure 7. Método Excluir

### 3.5. Método Jogar Partida

Para utilizar o método Jogar Partida, é necessário escolher a opção 5 no menu e já ter no banco 2 clubes criados. Após isso, é informado para o programa quais os times irão se enfrentar e o placar do jogo, da seguinte forma:

```
Opcao:
5
Aqui realizamos partidas entre dois times.
Entre com o id do primeiro clube:
1
Entre com o id do segundo clube:
2
PLACAR:
Vasco da Gama:
1
Atlético:
4
O clube Atlético ganhou! Ele faz 3 pontos e o clube Vasco da Gama 0 pontos!
```

Figure 8. Informações completas para método Jogar Partida

E para completar, é feita uma atualização com os novos valores de pontos e partidas jogadas de cada clube.

```
Nome: Vasco da Gama
CNPJ: 12345678
Cidade: Rio de Janeiro
Partidas Jogadas: 1
Pontos: 0
TIME ALTERADO

Nome: Atlético
CNPJ: 12345
Cidade: Belo Horizonte
Partidas Jogadas: 1
Pontos: 3
TIME ALTERADO
```

Figure 9. Informações atualizadas pós partida jogada

## 4. Conclusão

Podemos concluir que com esse trabalho, que desenvolver aplicações com memória secundária é muito interessante e necessário, pois garantimos consistência e persistência dos

dados. Fizemos um sistema de CRUD simples que poderia muito bem ser modificado para atender necessidades pessoais e no mercado de trabalho, em que uma empresa poderia ter por exemplo um sistema para gerenciamento de equipamentos, em que um certo funcionário pode realizar ações de criação, atualização, pesquisa e deletar certo equipamento. Então, por meio desse trabalho aprendemos as operações básicas em um sistema que vai armazenando cada dado em memória secundária. Outra vantagem de se trabalhar em memória secundária é que se por algum motivo ocorresse falta de energia durante a execução do programa nenhum dado seria perdido, já que quando for reiniciado será necessário apenas recarregar o arquivo. Por fim, ainda aprendemos muito sobre a linguagem Java em si, que nunca tínhamos usado antes. Java é uma linguagem que lida muito bem com o fluxo de arquivos, e é umas das linguagens mais usadas no mercado de trabalho atualmente, por isso é tão importante saber sua sintaxe e ter projetos com ela.

## 5. Parte 2

### 5.1. Arquivo de Índice

A medida que um sistema cresce se torna indispensável a utilização de um arquivo de índice, já que podemos aumentar exponencialmente a performance de nossa aplicação. Então, visando a performance, nosso arquivo de índice tem apenas 2 valores por registro: o id do registro e a posição em que esse registro se encontra no arquivo de dados original. E este novo arquivo é modificado de acordo com as alterações no arquivo original, ou seja: assim que acrescentamos um novo registro no arquivo de dados original também adicionamos no arquivo de índice o id do registro e a posição. Quando um registro é alterado e sua posição sofre modificação, também alteramos o valor da posição desse registro no índice. Um detalhe importante a se ressaltar: como não aproveitamos os espaços deixados por registros deletados ele está sempre ordenado. Também é importante dizer que esses registros só são apagados de fato do arquivo de índice quando ocorre a ordenação externa, que será abordada mais a frente.



Figure 10. Estrutura do arquivo de índice

### 5.2. Busca Binária

Se por si só a utilização de um índice sequencial já aumenta a performance, nem se compara com um índice que utiliza busca binária para pesquisa. Todas as pesquisas no nosso programa foram substituídas por pesquisas binárias no arquivo de índice, e para isso existe uma função que recebe um parâmetro (o id do registro desejado) e retorna um long (a posição em que esse registro se encontra no arquivo original). Pesquisa binária é um algoritmo simples e bem famoso, porém agora estamos lidando com pesquisa em um arquivo que é dividido em registro de 9 bytes (1 para o id e 8 para a posição), e portanto, foi necessário fazer algumas adaptações. O código ficou da seguinte maneira:

Nesse código tivemos que multiplicar e dividir por 9 em alguns momentos específicos, para que a leitura ocorra sempre na primeira posição do registro (a do id, que é o atributo que deve ser comparado).

### 5.3. Ordenação Externa

Para manter os dados sempre atualizados e de fácil acesso implementamos mais uma funcionalidade, a ordenação externa. Como nosso arquivo de índice estava sempre ordenado, decidimos implementar essa funcionalidade diretamente no arquivo de dados original. Isso porque após vários "updates" e "deletes" nosso arquivo fica com muitos espaços que podemos chamar de "lixo". Após a ordenação ocorrer todos esses "lixos" são retirados e nosso arquivo de dados volta a ficar ordenado e com todos os espaços com dados relevantes.

```

// pesquisa no arquivo de index a posição de um id
public long search(byte id) {
    try {
        RandomAccessFile arq = new RandomAccessFile(name: "dados/index.db", mode: "rw"); // abre o arquivo ou cria se ele não existir
        long low = 0, high = arq.length() / 9, mid; // davidson, há 6 dias • add inverted list ...
        byte idArq;
        // nosso index sempre estará ordenado, então podemos ir para o meio do arquivo e começar a busca binária
        while(low ≤ high) {
            mid = (int)((low + high) / 2);
            arq.seek(mid * 9);
            idArq = arq.readByte();
            if(id < idArq)
                high = mid - 1;
            else if(id > idArq)
                low = mid + 1;
            else {
                return arq.readLong();
            }
        }
        arq.close();
    } catch(Exception e) {
        e.printStackTrace();
    }
    long lixo = -1;
    return lixo;
}

```

**Figure 11. Código adaptado da pesquisa binária**

Essa ordenação tem alguns requisitos por conta do trabalho: deveríamos utilizar a intercalação balanceada com 2 caminhos e com a capacidade de ordenação de memória principal limitada a 10 registros. Em outras palavras: deveríamos ter 2 arquivos temporários para leitura e dois para escrita. E todos esses requisitos foram atendidos em nossa implementação.

O algoritmo funciona, basicamente em duas etapas: a distribuição e a intercalação. Em primeiro lugar vem a distribuição de todo o arquivo de dados em 2 arquivos temporários, sempre pegando de 10 em 10 registros (já que é o limite de ordenação em memória principal). Depois de ordenar esses 10 registros salvamos todos em sequência nos arquivos temporário de maneira alternada, ou seja, hora se salva no primeiro arquivo hora no segundo. Fazemos isso até que não haja mais registros no arquivo de dados. Após isso a distribuição está concluída e damos início a intercalação.

Na intercalação agora teremos 4 arquivos: 2 de leitura e 2 de escrita. então iremos carregar na memória 10 registros dos dois arquivos de leitura (aqueles que estavam montados na distribuição) e vamos escrevendo esses registros (de maneira ordenada) em um arquivo de escrita. Depois, repetimos o processo escrevendo no outro arquivo de escrita. Fazemos isso até que não haja mais registros nos arquivos de leitura. Após isso terminamos uma intercalação e iremos reiniciar o processo, só que dessa vez carregando 20 por vez. Depois faremos novamente carregando 40. E faremos isso até conseguir escrever todos os registros em apenas 1 arquivo de escrita. Contudo, o resultado será o arquivo completo sem a presença de "lixos".

A seguir segue um exemplo do arquivo de dados original antes de ser ordenado:

Como pode ser observado, o registro de nome "Galo" e id 1 foi substituído por um registro de nome "Atletico MG", portanto quando formos ordeanar devemos deixar apenas o último registro na primeira posição. Nossa ordenação ocorre sempre que o usuário seleciona o opção de sair do programa, e este arquivo ficaria da seguinte forma após esse processo:

Agora podemos ver que o registro com nome "Galo" foi apagado e restou apenas



00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
00000000 04 2A 00 00 00 13 01 00 04 47 61 6C 6F 00 05 31	. * . . . . . G a l o . . 1
00000010 32 33 34 35 00 02 42 48 00 00 20 00 00 00 16 02	2 3 4 5 . . B H . . . . .
00000020 00 07 41 6D 65 72 69 63 61 00 05 35 34 33 32 31	. . A m e r i c a . . 5 4 3 2 1
00000030 00 02 42 48 00 00 20 00 00 00 17 03 00 08 43 72	. . B H . . . . . C r
00000040 75 7A 65 69 72 6F 00 05 33 34 32 31 32 00 02 42	u z e i r o . . 3 4 2 1 2 . . B
00000050 48 00 00 20 00 00 00 17 04 00 08 46 6C 61 6D 65	H . . . . . F l a m e
00000060 6E 67 6F 00 05 35 31 32 33 34 00 02 42 48 00 00	n g o . . 5 1 2 3 4 . . B H . .
00000070 20 00 00 00 1A 01 00 0B 41 74 6C 65 74 69 63 6F	. . . . . A t l e t i c o
00000080 20 4D 47 00 05 31 32 33 34 35 00 02 42 48 00 00	M G . . 1 2 3 4 5 . . B H . .

Figure 12. Arquivo de dados antes de ser ordenado

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
00000000 03 20 00 00 00 1A 01 00 0B 41 74 6C 65 74 69 63	. . . . . A t l e t i c
00000010 6F 20 4D 47 00 05 31 32 33 34 35 00 02 42 48 00	o M G . . 1 2 3 4 5 . . B H .
00000020 00 20 00 00 00 16 02 00 07 41 6D 65 72 69 63 61	. . . . . A m e r i c a
00000030 00 05 35 34 33 32 31 00 02 42 48 00 00 20 00 00	. . 5 4 3 2 1 . . B H . . .
00000040 00 17 03 00 08 43 72 75 7A 65 69 72 6F 00 05 35	. . . . . C r u z e i r o . . 5
00000050 33 32 34 32 00 02 42 48 00 00	3 2 4 2 . . B H . .

Figure 13. Arquivo de dados depois de ser ordenado

o registro de nome "Atletico MG" com id 1 e lápide válida.

#### 5.4. Lista Invertida

E para finalizar as novas funcionalidades nessa etapa dois do trabalho adicionamos a pesquisa por nome do clube ou nome da cidade do clube.

Toda essa funcionalidade de pesquisa foi implementada utilizando a estrutura de uma lista invertida, em que temos a seguinte estrutura: a palavra (uma string), 4 bytes (cada byte representa um id) e um long (que é a posição do próximo id que possui a palavra, se existir). Sua estrutura no arquivo é a seguinte:

Palavra - BH Id's 1, 2, 3 Posição - -1 (Não existe)

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
00 02 42 48 01 02 03 FF FF FF FF FF FF FF FF	. . B H . .

Figure 14. Arquivo de dados depois de ser ordenado

O algoritmo de lista invertida funciona da seguinte maneira: nós vamos salvando as palavras dentro da estrutura mostrada acima, sempre colocando os id's apenas em posições vazias (preenchidas com -1). E quando houver já 4 id's para apenas 1 palavra iremos salvar uma posição que indica onde está o próximo id, dessa forma garantindo que

independentemente da quantidade de clubes com o mesmo nome será possível armazenar todos e relacioná-los entre si.

E a pesquisa funciona da seguinte maneira: o usuário digita uma string e nós separamos essas strings em um array de strings separados pelo "espaço". Ou seja, se o usuário digitar "Atletico MG", e supondo que nossa base dados original seja a seguinte:

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
05 20 00 00 00 1A 01 00 0B 41 74 6C 65 74 69 63	. . . . . A t l e t i c
6F 20 4D 47 00 05 31 32 33 34 35 00 02 42 48 00	o M G . . 1 2 3 4 5 . . B H .
00 20 00 00 00 19 02 00 0A 41 6D 65 72 69 63 61	. . . . . A m e r i c a
20 4D 47 00 05 35 34 33 32 31 00 02 42 48 00 00	M G . . 5 4 3 2 1 . . B H .
20 00 00 00 17 03 00 08 43 72 75 7A 65 69 72 6F	. . . . . C r u z e i r o
00 05 35 33 32 34 32 00 02 42 48 00 00 20 00 00	. . 5 3 2 4 2 . . B H . . .
00 1A 04 00 0B 41 74 6C 65 74 69 63 6F 20 50 52	. . . . . A t l e t i c o P R
00 05 34 35 32 33 35 00 02 50 52 00 00 20 00 00	. . 4 5 2 3 5 . . P R . . .
00 1A 05 00 0B 41 74 6C 65 74 69 63 6F 20 47 4F	. . . . . A t l e t i c o G O
00 05 32 36 33 34 36 00 02 47 4F 00 00	. . 2 6 3 4 6 . . G O . .

Figure 15. Arquivo de dados

Nesse caso, iríamos dividir a string em: ["Atletico", "MG"]. A partir daí faríamos a pesquisa para cada uma das strings no array, e a palavra "Atletico" retornaria os id's: [1, 4, 5], enquanto a palavra "MG" retornaria os id's: [1, 2]. Após isso fazemos a interseção entre os 2 conjuntos e teremos apenas o id 1, logo o resultado que será mostrado é apenas o clube de id 1, conforme pode ser visto abaixo:

```
Você entrou no método de pesquisar por nome do clube.
Entre com o nome do Clube:
Atletico MG
Atletico
MG
[1]
Os clubes que correspondem ao nome digitado são:

Nome: Atletico MG
CNPJ: 12345
Cidade: BH
Partidas Jogadas: 0
Pontos: 0
```

Figure 16. Resultado

Importante ressaltar apenas que temos um arquivo de lista invertida para a pesquisa por nome e um arquivo para a pesquisa por cidade, porém o funcionamento de ambos é exatamente igual.

## 5.5. Conclusão

Neste trabalho fomos além da parte 1 (que era simplesmente um CRUD básico) e implementamos funcionalidades muito relacionadas com a parte que o usuário não percebe

diretamente, mas que é muito importante para a performance e a funcionalidade do sistema como um todo. Aplicamos diversos conceitos que vimos apenas a lógica durante a aula, então todo o código é de total autoria e desenvolvimento do grupo.

Das funcionalidades aplicadas nesta parte 2 destacam-se: a pesquisa binária com arquivo de índice, que aumenta exponencialmente a velocidade de pesquisa por um registro (ainda mais em uma grande base de dados). Também tem a ordenação externa que acontece em memória secundária, já que dependendo da quantidade de dados a memória primária não daria conta de carregar todos os registros. Com essa ordenação garantimos que não há "lixo" em nossos arquivos de dados e de índice. E por fim, mas não menos importante, temos a lista invertida, que é um mecanismo de busca impressionante, pois não retorna apenas um registro, mas sim todos os registros relacionados a palavra (ou conjunto de palavras) que está sendo pesquisado. Isso tudo sempre visando o melhor desempenho possível.

## 6. Parte 3

### 6.1. Compressão LZW

Para o desenvolvimento da compressão LZW, foi necessário primeiramente a criação de um dicionário com os caracteres aceitos para a inclusão de um Clube. Como iremos salvar os dados do nome criptografados, nosso dicionário vai de 0 à 255, incluindo a grande maioria dos símbolos da tabela ASCII. Após isso já se inicia o método de compressão dos dados, que ocorre da seguinte forma, a palavra vai ser lida, sendo salva no novo arquivo e além disso incrementando as novas combinações no dicionário, fazendo com que se torne possível diminuir o número de caracteres em relação à palavra inicial.

Agora, para utilizar o método de compressão é necessário informar qual a versão do arquivo será gerada. Após isso será criado um arquivo chamado "ClubeCompressao" + número da versão gerada. Além de informar qual é a porcentagem de ganho que o arquivo teve ao ser comprimido, da seguinte forma:

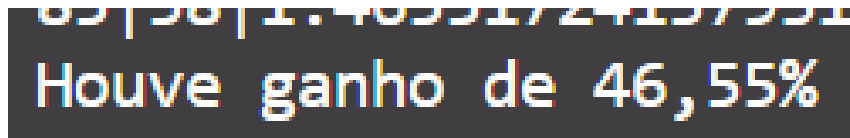


Figure 17. Perda/Ganho da compressão do arquivo

O novo arquivo se encontra da seguinte forma:

```
00000000 02 20 00 00 00 25 01 41 74 6C 65 74 69 63 6F 20 . . . . % . A t l e t i c o
00000010 4D 47 3B 31 32 33 34 35 3B 42 48 3B 00 00 20 00 M G ; 1 2 3 4 5 ; B H ; . . .
00000020 00 00 23 02 41 6D 65 72 69 63 61 20 4D 47 3B 35 . . # . A m e r i c a M G ; 5
00000030 34 33 32 31 3B 42 48 3B 00 00 4 3 2 1 ; B H ; . .
```

Figure 18. Resultado Arquivo Comprimido

### 6.2. Descompressão LZW

Além da compressão, foi criado um método para descomprimir o arquivo gerado anteriormente. Pelo qual você informa a versão em que deseja visualizar os dados descomprimidos e com isso é printado todas as informações e sobrescrevendo o arquivo Clube principal com os novos dados descomprimidos, da seguinte forma:

```
00000000 02 20 00 00 00 25 01 00 16 C3 81 C3 B4 C3 AC C3 . . . . % . . . . .
00000010 A5 C3 B4 C3 A9 C3 A3 C3 AF C2 A0 C3 8D C3 87 00 . . . . .
00000020 05 31 32 33 34 35 00 02 42 48 00 00 20 00 00 00 . 1 2 3 4 5 . . B H . . .
00000030 23 02 00 14 C3 81 C3 AD C3 A5 C3 B2 C3 A9 C3 A3 # . . . . .
00000040 C3 A1 C2 A0 C3 8D C3 87 00 05 35 34 33 32 31 00 . . . . . 5 4 3 2 1 .
00000050 02 42 48 00 00 . B H . .
```

Figure 19. Resultado Arquivo Descomprimido

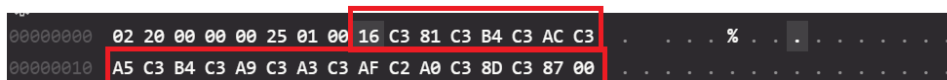
### 6.3. Criptografia de dados

Como orientado no TP o algoritmo usado na criptografia ficou à escolha do grupo. Nós utilizamos a cifra de Cezar, que é uma das primeiras e mais simples técnicas de criptografias conhecidas.

Ela é um tipo de cifra de substituição, em que cada letra do texto é substituída por outra que se apresenta no alfabeto abaixo dela um número fixo de vezes. Como esse número é fixo podemos ainda dizer que esse é um tipo de criptografia simétrica, já que utilizamos da mesma chave para cifrar e decifrar a mesma mensagem.

Nossa chave foi definida como 128, portanto ao cifrar somamos 128 a cada caracter do nome e ao decifrar subtraímos 128 de cada caracter. O único campo criptografado é o nome, portanto temos que tomar cuidado de sempre que alterarmos o campo nome no arquivo de dados usar o método para criptografar e para mostrar na tela é necessário utilizar o método para descriptografar o campo.

O campo nome com o valor de "Atlético MG" criptografado fica da seguinte maneira:



```
00000000 02 20 00 00 00 25 01 00 16 C3 81 C3 B4 C3 AC C3
00000010 A5 C3 B4 C3 A9 C3 A3 C3 AF C2 A0 C3 8D C3 87 00
```

Figure 20. Nome Atlético MG Criptografado

### 6.4. Conclusão

Por fim, podemos colocar na prática conceitos importantíssimos relacionados à performance e segurança dos dados de nosso sistema.

Com a compressão podemos diminuir(e muito) o tamanho do arquivo pelo qual armazena as informações do nosso sistema de CRUD. Com isso, podemos levar essa implementação para outros cenários, principalmente quando temos uma grande massa de dados e queremos diminuí-la.

Em relação a criptografia, foi outro ponto muito interessante de se trabalhar, pois garantimos certa segurança em informações sensíveis do usuário. Contudo, em um sistema real nosso algoritmo necessitaria sofrer algumas alterações, visto que a cifra de substituição por si só não é suficiente para garantir a segurança, já que é um algoritmo relativamente fácil de ser decifrado. O ideal seria utilizar algum algoritmo de maior eficiência, como os de chave assimétrica.