

## 1) Introdução

Tabelas Hash, do inglês Hash Table são estruturas de dados indicadas para aplicações que precisem de métodos rápidos de inserção, remoção e busca de dados, mas onde a ordenação não é importante. Numa boa implementação de tabela Hash, a complexidade dessas três operações é bastante reduzida, se compararmos com uma lista linear, onde só a inserção é feita com eficiência máxima.

A idéia por trás da tabela Hash é bem simples. Usando-se uma função de transformação, também chamada de Função Hash, codifica-se o dado a ser armazenado em um número, e usa-se esse número para indexar uma tabela. A idéia é, sempre que se precisar deste dado, ao invés de varrer toda uma tabela ou lista em busca do mesmo, fazendo comparações e iterações, procura-se pelo *HashCode*, o código gerado pela função Hash para aquele dado. Em geral isto é feito computando-se um valor numérico para aquele dado, e reduzindo-se esse valor *modulo* o tamanho desejado da tabela.

As coisas ficam um pouco mais complicadas quando em nossa tabela começam a aparecer valores duplicados. Um Hash perfeito gera um valor único para cada dado inserido, mas isso nem sempre é ideal ou viável, por isso muitas vezes ocorrem colisões, ou seja, vários dados diferentes geram um mesmo *hashCode* e portanto, em teoria, devem ser armazenados no mesmo lugar. Existem vários métodos para tratar esse tipo de problema, o utilizado nesta implementação trabalha com listas encadeadas.

Para tratar colisões com listas encadeadas, consideramos nossa Tabela Hash como um arranjo de listas encadeadas, e em cada nodo destas é que armazenaremos nossos dados. O custo deste tratamento é um aumento na complexidade para as buscas na tabela, pois ainda é necessário varrer listas encadeadas. O segredo para manter a eficiência razoável é gerar índices o mais esparsos e bem distribuídos possíveis, de forma que cada índice da tabela contenha o mínimo possível de nodos encadeados; por isso é essencial que a função de hash seja eficiente em gerar códigos com o mínimo de repetições possível.

Uma função hash simples, porém eficiente, para Strings consiste em atribuir ao hash um somatório dos valores ASCII de todos os caracteres da String, multiplicados um a um por um valor inteiro arbitrário, em geral um número primo, no final, obtém-se o módulo deste número pela quantidade de espaços na tabela. Essa é uma função hash bem rápida, o que é essencial, já que deverá ser chamada dentro de todas as operações de nossa tabela Hash, e relativamente eficiente, pois gera algo muito próximo do mínimo possível de colisões. Uma outra opção seria gerar Checksums criptográficos dos dados a inserir, usando um algoritmo como MD5, por exemplo, mas ao mesmo tempo que isso geraria um Hash perfeito, sem colisões, o MD5 é muito menos eficiente que nossa pequena função Hash, e executá-lo a cada chamada de função da tabela seria no fim muito menos eficiente do que aceitar algumas colisões.

Como dito anteriormente, porém, as tabelas Hash não preservam de forma alguma a ordenação das informações, elas são armazenadas quase que aleatoriamente na tabela, o que implica na necessidade de uma função extra, e com complexidade razoável, caso queiramos obter algum tipo de ordenação.

## 2) Estruturas e Tipos de Dados

A minha solução implementa a tabela Hash como um arranjo de listas encadeadas do tipo `HumanList`, o arranjo possui 13 índices, conforme especificado, e portanto a minha função Hash gera códigos variando de 0 a 12. Os dados serão gravados em objetos do tipo `HumanNode`, que representa o nodo de uma `HumanList`.

Quanto ao tratamento de colisões, optei por implementar uma lista duplamente encadeada, não genérica e sem nodo cabeça. Este último não se mostrou necessário, pela própria natureza das listas duplamente encadeadas, que foram escolhidas por oferecerem uma maior flexibilidade para inserção e menor complexidade de busca que as listas simplesmente encadeadas. Optei por implementar uma lista não genérica a título de simplicidade do entendimento, caso eu houvesse criado um nodo genérico para a classe `HumanList`, teria que aumentar o número de ponteiros significativamente, criando um campo em cada nodo que apontaria para outro tipo de dados mais exclusivo, o que prejudicaria muito a compreensão.

Cada nodo do tipo `HumanNode` consiste de dois conjuntos de campos, o primeiro é relativo ao encadeamento, e contém informações sobre as relações que possui com os outros nodos da lista, o segundo conjunto contém as informações solicitadas: nome, cpf, idade, sexo e endereço.

## 3) Implementação

Decidi implementar duas versões do programa, uma usando o modo gráfico e outra o modo texto. Ambos têm muito em comum, portanto a descrição que se segue é aplicável a ambos.

Meu programa implementa `TabelaHash`, usando listas encadeadas do tipo `HumanList`, contendo nodos do tipo `HumanNode`, para armazenar dados de pessoas obtidos através de um arquivo de entrada. Esses dados serão organizados em duas instâncias diferentes de `HashTable`, sendo uma indexada por CPF e outra indexada por NOME

### Entrada:

A entrada consiste num arquivo com o seguinte formato:

```
nome1
endereço1
cpf1
idade1
sexo1
nome2
endereço2
cpf2
idade2
sexo2
endOfFile
```

Ao final do arquivo de entrada, para finalizar a entrada de dados, haver uma linha com palavra `endOfFile`, assim o programa pára a leitura e inicia o prompt para consulta de dados.

## Verificação:

### O Método de Hash:

Escolhi utilizar um método Hash bastante simples, porém também eficiente. Seu código Donte encontra-se a partir da linha 43 do arquivo TabelaHash.java

Basicamente este método recebe uma string qualquer como parâmetro, e a varre, caractere a caractere, somando ao resultado o valor da multiplicação do valor ASCII daquele caractere por um número primo arbitrário, em seguida, um novo número é gerado, multiplicando-se o primo original por outro arbitrário. No fim, para encaixarmos no limite de 13 espaços especificado para a tabela hash a ser implementada, fazemos um mod13. No fim, obtemos hashes variando de 0 a 12, e suficientemente esparsos para evitar muitas colisões.

A principal vantagem deste método sobre um método criptográfico por exemplo, é a sua velocidade. Um algoritmo criptográfico como o MD5, gera hashes sempre únicos, o que seria um Hash Perfeito, porém ele é um algoritmo mais lento, e chamá-lo a cada vez que fosse necessário obter um hash poderia diminuir consideravelmente a eficiência de nossa implementaç.

### Tratamento de Colisões:

Como já dito, escolhi implementar as colisões por encadeamento usando a lista que implementei, chamada HumanList. O Método de tratamento dessas colisões com o uso dessa lista é bem simples.

O método coração dessa implementação é o Insertrecord, na linha 15 do arquivo TabelaHash.java. Esse método recebe como parâmetros as informações obtidas da entrada, em seguida cria um novo nodo do tipo HumanNode utilizando essas informações.

O próximo passo é inserir na tabela hash os dados, para isso são feitas duas chamadas. Primeiramente é obtido o hash da string referente ao nome, e o nodo é inserido na tabela de nomes, indexado por este campo. A seguir, é obtido o hash do CPF, e o nodo é inserido na tabela de CPFs, indexado pelo campo devido.

Já para obter registros da tabela, foram precisos dois métodos diferentes, um que busca pelo CPF e outro que busca pelo nome.

Ambos possuem em comum a característica de obter logo no início o hash do campo que querem buscar, em seguida, cada um chama um método correspondente ao tipo de dado que pretendem retornar, dessa vez na classe HumanList. A busca é feita usando-se o resultado do Hash para saber em que posição da tabela, ou seja, em qual lista encadeada, deve ser feita a busca.

Os métodos chamados ( `getByName()` e `getByCPF()` ), fazem a busca dos dados na lista encadeada e retornam os dados corretos.

No caso da busca por CPF, apenas um nodo é retornado, pois o CPF é sempre único. Já no caso dos nomes, é retornada uma nova lista do tipo HumanList, contendo apenas os valores que satisfazem a busca.

### Pesquisas:

O motor de pesquisas é bastante simples, consistindo dos métodos `getByName` e `getByCPF` da classe HumanList. Ambos varrem a lista toda (portanto possuem complexidade  $O(n)$ , com  $n$  = tamanho da lista) em busca de valores que coincidam com a chave. Como os valores devem coincidir perfeitamente (um requisito do método `equals()` da classe String), não é possível pesquisar por partes de palavras, ou palavras chave. O CPF e o Nome devem ser digitados exatamente como constam no arquivo de entrada. A única exceção é com relação à caixa das letras. Meu método de Hash transforma todas as letras para caixa-baixa antes de calcular o hash, portanto, o programa encontra o valor desejado ignorando letras maiúsculas e minúsculas.

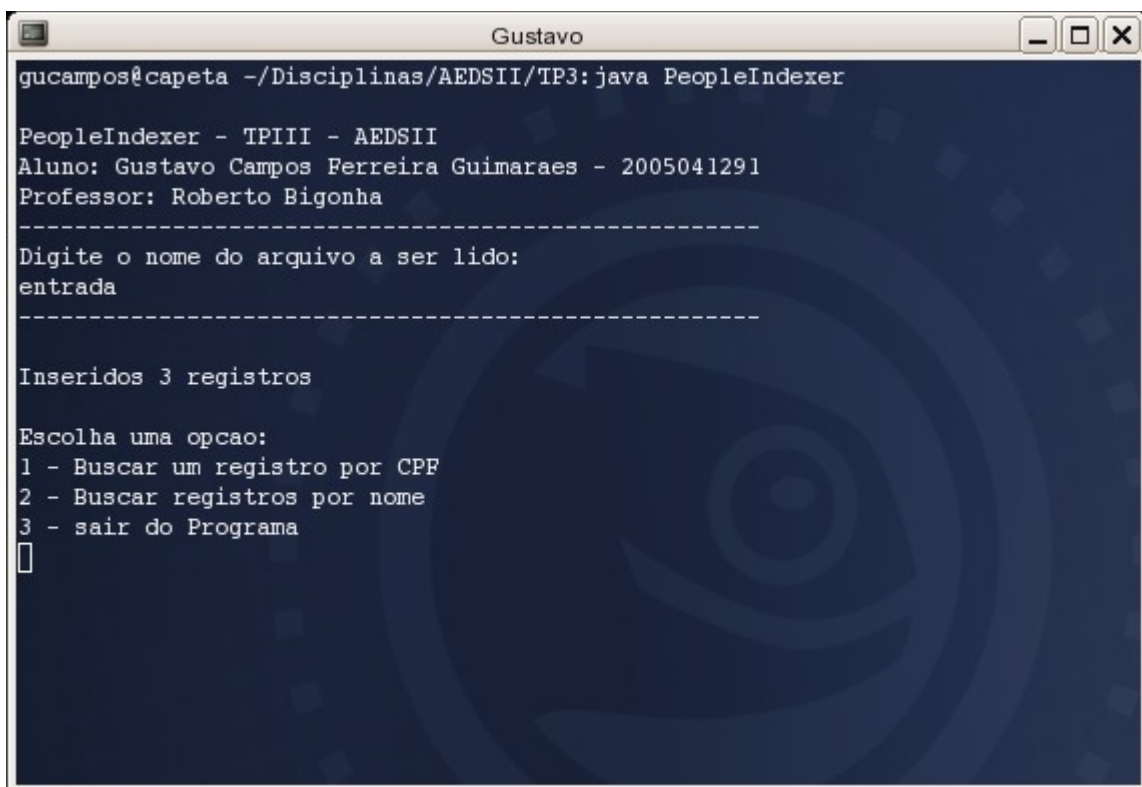
### A Versão Gráfica - GPeopleIndexer:

Uma aplicação como esta, de pesquisa e retorno de dados, é muito desconfortável se implementada de forma linear, como exige o modo texto dos sistemas operacionais. Receber entradas apenas do teclado torna a usabilidade muito demorada e complicada. Por isso, após finalizar a implementação de PeopleIndexer.java, resolvi tentar uma implementação gráfica, e após ser bem sucedido, fiquei bastante satisfeito com o resultado obtido.

A versão gráfica do programa basicamente difere da versão textual apenas nos métodos de entrada e saída. Ambos recebem um arquivo de entrada:

- A versão texto abre um prompt para que o nome do arquivo seja digitado
- A versão gráfica recebe o arquivo como parametro de linha de comando.

Quando às buscas, a versão texto mostra o seguinte prompt para realização da busca:



```
Gustavo
gucampos@capeta ~/Disciplinas/AEDSII/TP3: java PeopleIndexer

PeopleIndexer - TPIII - AEDSII
Aluno: Gustavo Campos Ferreira Guimaraes - 2005041291
Professor: Roberto Bigonha
-----
Digite o nome do arquivo a ser lido:
entrada
-----

Inseridos 3 registros

Escolha uma opcao:
1 - Buscar um registro por CPF
2 - Buscar registros por nome
3 - sair do Programa
█
```

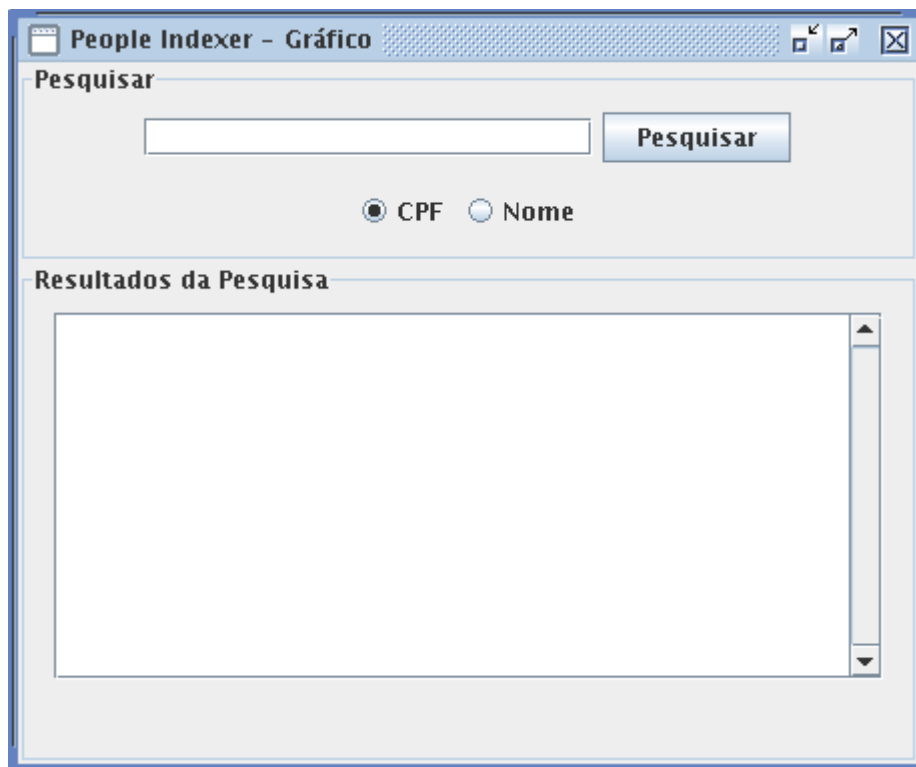
Nesta tela, digitando-se a opção pedida, será requisitada a digitação da chave a ser buscada, e em seguida os valores serão impressos na tela. É uma implementação bastante intuitiva e funcional, mas pode ser melhorada.

A versão gráfica usa a biblioteca Swing do Java para desenhar na tela elementos de interface com o usuário (UI), como botões e radio-buttons.

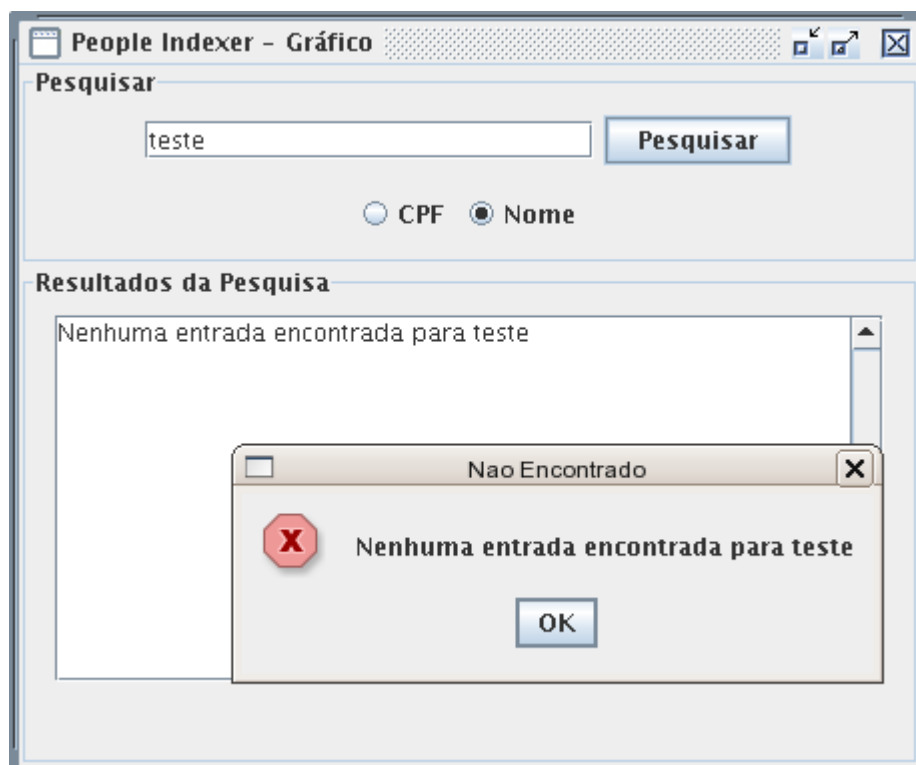
Para fazer uma busca nessa versão, basta escolher o índice clicando numa das RadioButtons, digitar o que se deseja buscar, e clicar em pesquisar. No painel abaixo serão retornados os resultados da busca. Caso algum erro aconteça, uma janela pop-up com informações do erro é mostrada.

A implementação gráfica tem a seguinte interface:

A interface do programa imediatamente após ser aberto.



Janela de aviso: nenhuma ocorrência encontrada



Resultados retornados: ignorando a caixa das letras

**People Indexer - Gráfico**

**Pesquisar**

gustavo **Pesquisar**

☐ CPF ☒ Nome

**Resultados da Pesquisa**

Ocorrencia 1

Nome: Gustavo  
CPF: 07705060620  
Idade: 20  
Sexo: m  
Endereço: Rua Sao Miguel 479, Bairro Itapoa

Ocorrencia 2

Nome: gustavo  
CPF: 123654789

## 4) Anexos

Estão incluídos dois anexos neste documento.

Anexo I – Código Fonte: contém o código fonte de todas as classes implementadas, sendo elas:

- HumanNode.java – A classe que implementa os nodos fundamentais
- HumanList.java – A classe que implementa a lista duplamente encadeada usada para armazenar as informações fazendo o tratamento de colisões
- TabelaHash.java – A classe que implementa a tabela Hash
- PeopleIndexer.java – O programa que utiliza as classes acima, modo texto
- GpeopleIndexer.java – O programa que utiliza as classes acima, modo gráfico

Anexo II – Métodos Implementados: contém uma listagem de todos os métodos das classes fundamentais, com seus cabeçalhos e uma breve descrição. (gerado com o uso do javadoc)