# Relatório Sobre o Trabalho de Implementação que Compõe a Segunda Nota da Disciplina Estrutura De Dados II

**João Marcello Mendes Moreira**

**e-mail: joaomarcello.mm@gmail.com**

**Professor: João Dallyson Sousa de Almeida**

**ESTRUTURA DE DADOS II**

**Resumo.** O vetor *hash* (ou tabela *hash*), é uma estrutura de dados que associa chaves de um dado a um índice em um vetor. Com isso, consegue fazer pesquisas rápidas chegando ao tempo O (1) no melhor caso. No trabalho de implementação, pediu-se a criação de um índice invertido utilizando-se um vetor *hash*. Neste relatório, falo um pouco sobre o índice invertido, explico o funcionamento do código e ainda apresento algumas comparações em relação aos diferentes tipos de tabela *hash* utilizadas na implementação.

## 1. Introdução

O vetor *hash* (ou tabela *hash*), é uma generalização de um vetor com *m* posições. Cada posição na tabela representa um endereço. Os elementos a serem armazenados nele possuem um valor-chave que é utilizado para calcular o endereço na tabela onde serão alocados (Ascencio e Araújo, 2010). A principal vantagem das tabelas *hash* é sua velocidade de pesquisa. No pior caso é equivalente a uma lista chegando a O (*n*), mas chega a O (1) no melhor caso.

Já o índice invertido é uma estrutura de dados que mapeia termos às suas ocorrências em um documento ou um conjunto de documentos. É uma estratégia de

indexação que permite a realização de buscas precisas e rápidas, em troca de maior dificuldade no ato de inserção e atualização de documentos (Wikipédia).

## 1.1 Motivação

Buscas por termos em uma lista tradicional exigiram percorrer cada documento e cada palavra dentro destes em busca do termo, enquanto que, com o uso de um índice invertido, pode-se saltar diretamente para o termo buscado. Logo, o uso deste recurso permite que os resultados sejam obtidos de forma consideravelmente mais rápida (Wikipédia).

O uso de listas invertidas tem o potencial de deixar as buscas mais eficientes, dado que estas permitem que sejam armazenadas informações adicionais que, acompanhadas de algoritmos adequados, tornam fácil a classificação e ordenação dos resultados. Ferramentas de busca como Google e Bing! utilizam como base índices invertidos para uma coleção de documentos.

## 1.3 O problema

Como atividade, pedia-se a implementação de um programa que produzia um índice invertido recebendo como entrada N arquivos de dados. Este programa deveria ser capaz de:

- Adicionar ao índice invertido somente as palavras com um tamanho maior que um valor C determinado pelo usuário.

- O índice invertido devia ser criado nas seguintes versões do TAD dicionário: *Hashing* Fechado (Endereçamento Aberto) com tratamento de colisões por tentativa linear e quadrático, *Hashing* Fechado com tratamento de colisões usando uma Árvore Rubro-Negra. Para efeito de comparação, a função de *hash* deveria ser implementada usando o método da divisão e da multiplicação.

- Analisar o desempenho de cada uma das versões do TAD para diferentes cenários.

- Utilizar o índice invertido construído utilizando um dos TAD's para realizar consultas de um ou mais termos. Dada uma consulta, o objetivo é retornar uma lista de documentos, ordenados (em ordem decrescente) pela relevância para a consulta.

- Imprimir o índice invertido: imprime as palavras em ordem alfabética, uma por linha. À frente de cada palavra, você deve imprimir a lista das ocorrências, ordenada pelo índice do documento. Para cada elemento da lista, você deverá imprimir o nome do documento (e não o identificador) e o número de ocorrências da palavra no documento.

## 2. Implementação

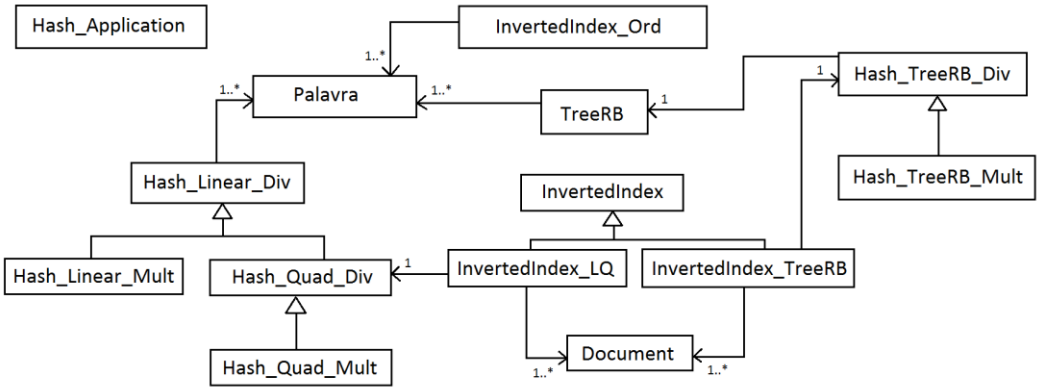
Para a implementação do programa, foi utilizada a linguagem Java e a IDE NetBeans 8.2. A seguir apresento a funcionalidade das classes do programa assim como a estrutura do mesmo.

## 2.1 Classes utilizadas no programa

O programa foi dividido nas seguintes classes: *Document*, *Hash_Application*, *Hash_Linear_Div*, *Hash_Linear_Mult*, *Hash_Quad_Div*, *Hash_Quad_Mult*, *Hash_TreeRB_Div*, *Hash_TreeRB_Mult*, *InvertedIndex*, *InvertedIndex_Ord*, *InvertedIndex_LQ*, *InvertedIndex_TreeRB*, *Palavra* e *TreeRB*. A organização e composição das classes está ilustrada na figura 2.1.

Foi utilizada ainda o pacote *sortMethods* (construído na primeira atividade de implementação da disciplina) que contém classes que realizam a ordenação de um vetor de elementos do tipo *KeyElement*.

**Figura 2.1 Organização e composição das classes do programa**




Fonte: próprio aluno.

A seguir, apresento uma breve descrição do funcionamento de cada uma das classes.

### 2.1.1 Document

Guarda as informações de um determinado documento (o caminho, a relevância do documento para uma consulta e a chave do documento, que é gerada automaticamente pelo programa). Como em algum momento do programa um vetor de *Document* precisará ser ordenado, implementa a interface *KeyElement*.

### 2.1.2 Palavra

Representa as informações de uma palavra encontrada em um documento. A chave é gerada somando-se os valores da tabela ASCII de cada caractere. Possui uma classe interna *Doc* que guarda o id de um documento e a quantidade de vezes que essa palavra aparece no documento. Um *arrayList* de *Doc* é utilizado pela classe para guardar todas as ocorrências dessa palavra em um conjunto de documentos. Como em algum momento do programa um vetor de Palavra precisará ser ordenado, implementa a interface *KeyElement*.

### 2.1.3 Hash_Application

Contém a função main.

### 2.1.4 Hash_Linear_Div

Implementa uma tabela *hash* que trata as colisões de forma linear e usa o método da divisão como função *hash*. É a superclasse das classes *Hash_Linear_Mult* e *Hash_Quad_Div*. Como o *hash* e do tipo endereçamento aberto, pode chegar a situação a qual um elemento não pode ser inserido não possuir espaço na tabela. Por isso, fez-se necessário criar uma função chamada *aumentarTamanho* que, como o nome sugere, aumenta o tamanho da tabela *hash*.

### 2.1.5 Hash_Linear_Mult

Implementa uma tabela *hash* que trata as colisões de forma linear e usa o método da multiplicação com função *hash*. É subclasse de *Hash_Linear_Div*.

### 2.1.6 Hash_Quad_Div

Implementa uma tabela *hash* que trata as colisões de forma quadrática e usa o método da divisão com função *hash*. É subclasse de *Hash_Linear_Div*.

### 2.1.7 Hash_Quad_Mult

Implementa uma tabela hash que trata as colisões de forma quadrática e usa o método da multiplicação com função hash. É subclasse de Hash_Quad_Div.

### 2.1.8 Hash_TreeRB_Div

Implementa uma tabela *hash* que trata as colisões com uma árvore rubro negra e usa o método da divisão com função *hash*.

#### 2.1.9 Hash_TreeRB_Mult

Implementa uma tabela *hash* que trata as colisões com uma árvore rubro negra e usa o método da multiplicação com função *hash*. É subclasse de *Hash_TreeRB_Div*.

#### 2.1.10 InvertedIndex

Interface que especifica as funções que uma estrutura índice invertido precisa implementar.

#### 2.1.11 InvertedIndex_LQ

Classe que implementa *InvertedIndex*. Usa uma instância de *Hash_Linear_Div* ou *Hash_Linear_Mult* para as suas operações.

#### 2.1.12 InvertedIndex_TreeRB

Classe que implementa *InvertedIndex.* Usa uma instância de *Hash_TreeRB_Div* ou *Hash_TreeRB_Mult* para as suas operações.

#### 2.1.13 InvertedIndex_Ord

Classe usada para imprimir o índice invertido. Basicamente, ela pega todas as Palavra que estão em um *Hash_Linear_Div* ou *Hash_TreeRB_Div* e guarda em um vetor de Palavra. Com isso, usa a classe *Ordenator* do pacote *sortMethods* para realizar a ordenação das Palavras do vetor em ordem alfabética.

### 2.2 Funcionamento do programa

É solicitado ao usuário algumas informações que serão necessárias para a criação do *hash*. Primeiramente, é preciso informar a quantidade *N* de arquivos de texto utilizados (os arquivos estão na mesma pasta do programa com o nome de “doc1.txt”, “doc2.txt”,

“doc3.txt” e “doc4.txt”). Se algum arquivo informado não puder ser aberto, o programa finaliza a sua execução.

Em seguida, é solicitado o caminho para esses *N* arquivos. Depois, o usuário deve informar o modo como a tabela *hash* deve tratar as colisões (“linear”, “quad” ou “tree”) e a função de *hash* utilizada para mapear a posição do elemento na tabela (“div” ou “mult”). É solicitado ainda um inteiro *C* que serve para limitar o tamanho das palavras que serão adicionadas na tabela *hash*. Palavras menores que *C* não serão incluídas.

Com todas essas informações, o programa enfim cria uma tabela *hash* de acordo com as informações fornecidas pelo usuário. Em seguida, um menu aparece com as seguintes opções: Realizar consulta, imprimir índice invertido e sair. Caso a opção escolhida seja realizar uma consulta, o usuário deve digitar uma ou mais palavras que ele deseja consultar nos documentos. O programa mostra então uma lista de documentos ordenados por relevância dada essa consulta, mas somente os documentos cuja relevância seja maior que um limiar estabelecido pelo programa (no caso 0,03). Já se opção for imprimir índice invertido, o programa mostra todas as palavras diferentes encontradas nos documentos e, ao lado de cada palavra, uma lista com o caminho do documento em que se encontra a palavra e a quantidade de vezes em que a palavra aparece nesse documento. Por fim, se a opção escolhida for sair, o programa finaliza a sua execução.

## 3. Testes

Para os testes, foi considerado o tempo que o programa demorava para construir o índice invertido e a quantidade de colisões que ocorreram nesse processo. Foram utilizados os mesmos arquivos para todos os tipos de tabela *hash* implementados.

**Tabela 3.1 Comparação entre as tabelas *hash* usando o método da divisão como função *hash***

| Tratamento de Colisão | Tempo para criar Índice Invertido (ms) | Quantidade de colisões |
|---|---|---|
| Linear | 47 | 3623 |

| | | |
|---|---|---|
| Quadrático | 47 | 3593 |
| Árvore RB | 31 | 2029 |

Fonte: próprio aluno.

**Tabela 3.2 Comparação entre as tabelas *hash* usando o método da multiplicação como função *hash***

| Tratamento de Colisão | Tempo para criar Índice Invertido (ms) | Quantidade de colisões |
|---|---|---|
| Linear | 62 | 4306 |
| Quadrático | 47 | 3061 |
| Árvore RB | 46 | 2029 |

Fonte: próprio aluno.

Em relação ao tempo, não foi percebido nenhuma grande diferença, provavelmente por conta do tamanho e a quantidade dos arquivos que, a princípio, eu havia julgado já serem consideráveis. Mas pudemos perceber um fato curioso em relação a quantidade de colisões da tabela *hash* que utiliza a árvore rubro negra para tratamento de colisões: é bem menor que a quantidade dos outros tipos de tabela *hash*. Isso se deve ao fato de que, em um *hash* com árvore RB, a colisão só acontece uma única vez para um elemento que cause colisão, enquanto que nos outros tipos de *hash*, há colisão enquanto não achar uma posição para colocar o elemento. Ainda, numa tabela *hash* com endereçamento aberto, o tamanho da tabela pode ser um problema, visto que pode faltar espaço para se adicionar um elemento. Logo, é necessária a criação de uma nova tabela *hash* em que os elementos da primeira tabela serão copiados para a mesma. Nesse processo, acontecem novas colisões que foram contabilizadas no teste, aumentando assim a quantidade de colisões.

## 4. Conclusão

Nesse trabalho, puder desenvolver uma interessante aplicação que usa as vantagens da tabela *hash* a seu favor. Consistia em simular os mecanismos de pesquisa que ferramentas de busca como o Google e Bing usam na sua implementação. No

processo de implementação do algoritmo, pude ter um conhecimento bem mais profundo em relação as estruturas de dados estudados em sala de aula (tabela *hash* e árvore rubro negra) e também pude sentir a dificuldade em implementá-las.

Falando em dificuldades, a minha principal foi, sem dúvidas, em relação à implementação da árvore rubro negra. Apesar de fazer exatamente como nos livros (com algumas adaptações para se encaixar na minha implementação), para arquivos muitos grandes, por algum motivo que eu não consegui perceber, algumas palavras apareciam duplicadas quando se imprimia o índice invertido.

## Referências

CORMEN, T. H. et al. **Algoritmos Teoria e Prática.** Tradução da segunda edição [americana] Vandenberg D. de Souza. - Rio de Janeiro: Elsevier, 2002.

ASCENCIO, A. F. G., ARAÚJO, G. S. **Estrutura de dados – algoritmos, análise da complexidade e implementações em JAVA/C++.** São Paulo: Pearson, 2010.

WIKIPÉDIA. **Listas invertidas – Wikipédia, a enciclopédia livre**. Disponível em: </https://pt.wikipedia.org/wiki/Listas_invertidas />. Acessado em: 26 de outubro de 2018.

WIKIPÉDIA. **Tabela de dispersão – Wikipédia, a enciclopédia livre.** Disponível em: </https://pt.wikipedia.org/wiki/Tabela_de_dispers%C3%A3o/>. Acessado em: 26 de outubro de 2018.

Notas de aula do professor João Dallyson Sousa de Almeida, UFMA.