Indexação de Notícias

Luis Augusto Toscano Guimarães
ltoscano@ice.ufjf.br

Universidade Federal de Juiz de Fora
Programa de Pós Graduação em Ciência da Computação
Juiz de Fora - MG, Brasil

1 Problema

O problema deste trabalho pode ser dividido em dois outros problemas, sendo eles a criação de um índice invertido e o cálculo de relevância dos elementos indexados por este índice. O índice invertido é uma estrutura onde cada entrada é representada por uma palavra que está contida em pelo menos um documento e, além disso, cada entrada do índice é associada a pares (id, count), onde "id" é um identificador de documento e "count" é a quantidade de vezes em que essa palavra aparece no documento em questão. Como exemplo, suponha 2 documentos com identificadores "doc1" e "doc2" e o conteúdo destes documentos como a seguir:

- doc1 = Quem quer casa. Porém ninguém casa. Ninguém quer casa também. Quer apartamento.
- doc2 = Ninguém em casa. Todos saíram. Todos. Quer entrar? Quem? Quem?

Assim, o índice invertido construído para este exemplo será:

- apartamento (1, 1)
  casa (1, 4) (2, 1)
  em (2, 1)
  entrar (2, 1)
  ninguem (1, 2) (2, 1)
  porem (1, 1)
  quem (1, 1) (2, 2)
  quer (1, 3) (2, 1)
  sairam (2, 1)
  tambem (1, 1)
  todos (2, 2)

Construído o índice, queremos utilizá-lo para a realização de consultas. As consultas se constituem como uma lista de termos que devem ser buscados no índice para que seja retornada uma lista com os documentos mais relevantes para estes termos. Vários métodos estão disponíveis para o processo de estimativa de relevância dos documentos. Para este trabalho, o método utilizado será baseado na frequência dos termos da consulta em cada documento (*TF - Term Frequency*) e na frequência inversa dos documentos (*IDF - Inverse Document Frequency*). O *IDF* atua como

discriminador entre os documentos que são relevantes e os que não são. Um termo que aparece em muitos documentos resultará em um *IDF* baixo e, portanto, não é um bom discriminador, enquanto que um termo que não aparece em muitos documentos resultará em um *IDF* alto e, portanto, é um bom discriminador. A relevância *r(i)* para um determinado documento "i" será dada por:

$$r(i) = \frac{1}{n_i} \sum_{j=1}^{q} w_j^i$$

Onde "n_i" é o número de termos distintos de "i" e "w_j^i" é o peso do termo t_j no documento "i" que é dado por:

$$w_j^i = f_j^i \frac{log(N)}{d_j}$$

Onde f_j^i é o número de ocorrências do termo t_j no documento "i", d_j é o número de documentos na coleção que contém o termo t_j e "N" é o número de documentos na coleção. Caso o termo t_j não apareça no documento "i", $f_j^i = 0$. Fazendo o cálculo de relevância para o exemplo anterior e utilizando como consulta "quer todos", temos:

$$w_1^1 = 3\frac{log(2)}{2} = 1,5$$

$$w_2^1 = 1\frac{log(2)}{2} = 0,5$$

$$w_1^2 = 0\frac{log(2)}{1} = 0,0$$

$$w_2^2 = 2\frac{log(2)}{1} = 2,0$$

$$r(1) = \frac{1}{7}(1,5 + 0,0) \approx 0,2142857$$

$$r(2) = \frac{1}{8}(0,5 + 2,0) = 0,3125$$

Desta forma, pelos cálculos de relevância realizados, a lista de resultados é dada por doc2 e doc1, necessariamente, nesta ordem.

2 Sistema de indexação

Este trabalho consiste no desenvolvimento de um sistema de indexação para um conjunto de notícias obtidas do HuffPost entre 2012 e 2018 [1], totalizando mais de 200 mil registros. Estes registros são compostos de título, descrição, autores, data e URL da notícia. O sistema desenvolvido deverá:

- Permitir que o usuário faça consultas de até 2 termos.
- Retornar a lista de resultados ordenados pela relevância dos documentos.
- A lista de resultados deverá estar limitada a 20 resultados, podendo aparecer documentos que não possuem um dos termos, caso estes fiquem dentro dos 20 mais relevantes.
- Implementar duas estruturas de dados diferentes para a estrutura de índice, sendo elas: Tabela Hash, Árvore AVL, SkipList e Trie.
- Ser testado com 10 mil consultas aleatórias com um termo e 10 mil consultas aleatórias com dois termos.

[1] https://www.kaggle.com/rmisra/news-category-dataset

2.1 Abordagens utilizadas

O sistema desenvolvido neste trabalho consiste de um sistema web implementado utilizando a linguagem Java com JDK 1.8, além ter sido feito o uso do *Spring Framework*. O projeto foi criado através do Spring Boot, encapsulando todos os componentes e configurações necessários. Desta forma, até mesmo o servidor web utilizado, que foi o *Apache Tomcat*, está encapsulado no projeto. A IDE de desenvolvimento foi a *Netbeans*.

O primeiro ponto a ser descrito sobre este sistema é a modelagem de alguns elementos. A classe "*dev.ltoscano.indexer.model.News*" representa uma notícia que é lida do arquivo de dataset de notícias. Esta classe contém os campos de título, descrição, autores, data e URL da notícia e, além disso, também contém um mapa de palavras e suas frequências, sendo este mapa construído pelo método "*buildWordFrequencies()*" da própria classe.

O método "buildWordFrequencies()" faz uso da classe "*dev.ltoscano.indexer.token.Tokenizer*", para o processo de tokenização da notícia. Este processo consiste na obtenção das palavras (*Tokens*) contidos no título e na descrição da notícia e, por fim, a frequência destas palavras são obtidas com o uso de "*Collections.frequency()*". É importante destacar que ocorre um tratamento destas palavras contidas na notícia, sendo removido todos os caracteres que não são números ou letras, além de torná-las minúsculas. Assim, o sistema considera como palavra, qualquer sequência composta por apenas letras, apenas números ou uma combinação destes tipos de caracteres.

Outra modelagem importante para ser destacada é em relação às estruturas que compõem o índice de notícias. Cada entrada do índice é representada pela classe "*dev.ltoscano.indexer.model.IndexEntry*", que contém um termo (Palavra) e uma lista de documentos que contém este termo. Os documentos são representados pela classe "*dev.ltoscano.indexer.model.Document*", que contém o identificador de uma notícia e o peso do termo no documento (w_j^i). Em outras palavras, tomando o exemplo da seção anterior, a classe *IndexEntry* representa "apartamento (1, 1)" e a classe *Document* representa "(1, 1)", porém, ao invés de armazenar o identificador e a frequência do termo, já armazena o peso do termo no documento. Com as entradas construímos o índice que é implementado sobre uma estrutura de tabela Hash, uma árvore AVL, uma SkipList ou uma Trie.

A classe abstrata "*dev.ltoscano.indexer.structure.IndexStructure*" define os métodos que qualquer estrutura de dados que implementa o índice deve conter, por exemplo, a inserção, a busca, a verificação se o índice é vazio ou se determinada chave está contida nele. A classe "*dev.ltoscano.indexer.index.Index*" implementa o índice de notícias, fazendo uso de uma estrutura de dados para este índice. Logo que um objeto da classe é instanciado, o dataset de notícias é lido linha a linha, onde cada linha passa pelo parser JSON (*Gson*) e então é criado um objeto da classe *News*. O mapa de termos e frequências é obtido do objeto *News* para que sejam criados os objetos *Document* que serão colocados em uma entrada de índice. Após toda a construção de entradas de índice, os pesos dos termos nos documentos são calculados pelo método "calculateDocumentWeights()" e armazenados nos objetos *Document*, terminando o processo de construção do índice.

A classe *dev.ltoscano.indexer.index.Index* também contém o método "query()" que realiza a consulta sobre o índice de notícias. Essa consulta ao índice se baseia em três processos: Obtenção da entrada de índice referente ao termo buscado (Se existir), cálculo de relevância dos documentos e ordenação do resultado. A complexidade para obtenção da entrada de índice depende da estrutura de dados que está sendo usada para o índice, ou seja, pode ser a complexidade de busca de uma tabela Hash, uma árvore AVL, uma SkipList ou de uma Trie. Já a complexidade para o cálculo de relevância dos documentos varia de acordo com o número de documentos que contém aquele termo e, por fim, a complexidade da ordenação do resultado depende do algoritmo de ordenação utilizado. Neste trabalho, as estruturas de dados implementadas para o índice foram uma tabela hash, uma árvore AVL e uma Trie. Em relação ao algoritmo de ordenação, apenas um algoritmo foi implementado, sendo ele o *MergeSort* que está implementado na classe "*dev.ltoscano.indexer.sort.MergeSort*".

2.2 Estruturas implementadas para o índice

Como já dito anteriormente, as estruturas de dados que foram implementadas neste trabalho foram a tabela Hash, a árvore AVL e uma Trie R-way, que estão implementadas nos pacotes "*dev.ltoscano.indexer.structure.HashTable*", "*dev.ltoscano.indexer.structure.AVL*" e, por fim, "*dev.ltoscano.indexer.structure.Trie*".

A tabela hash implementada faz uso de uma estrutura de nó que armazena uma chave, um objeto *IndexEntry* e um outro nó, pois a implementação faz uso de encadeamento externo em caso de colisão. Em caso de colisão em uma determinada entrada da tabela, a inserção nesta lista encadeada é feita sempre no início da lista, não sendo necessária a verificação se o elemento já existe na lista, pois a chave do nó, é a palavra do *IndexEntry*, ou seja, ela já é única, pois não há dois objetos *IndexEntry* que representam a mesma palavra. Desta forma, este tipo de inserção foi pensado para melhorar o desempenho da tabela nas inserções.

A tabela hash também faz o redimensionamento automático de seu tamanho, caso o seu fator de carga ultrapasse um determinado limiar (0.75). Para este redimensionamento, uma nova tabela é criada com o dobro do tamanho anterior e os itens armazenados na tabela antiga são inseridos na nova tabela, este processo está implementado no método "rehashing()" da classe *HashTable*. O objetivo deste redimensionamento é a redução nas colisões, melhorando os tempos das operações da estrutura.

A função de hash que a tabela faz uso é do tipo "*Polynomial rolling hash*" [2] que é dada por:

$$hash(s) = \sum_{i=1}^{n-1} s[i]p^i mod(M)$$

Onde "s" é uma string, "s[i]" é o caracter na posição "i" da string, "p" é um número primo e M seria o tamanho da tabela. É importante destacar que foi necessário um cuidado na implementação desta função de hash para que o cálculo deste somatório fosse realizado. Este somatório nada mais é do que um polinômio do tipo $s[0] + s[1]p + s[2]p^2 + ... + s[n-1]p^{n-1}$. Caso a implementação seja feita da seguinte forma:

```
1  public int hash(String s, int M)
2  {
3      int p = 31;
4      int sum = 0;
5
6      for(int i = 0; i < s.length(); i++)
7      {
8          sum += s.chartAt(i) * Math.pow(p, i);
9      }
10
11     return (sum % M);
12 }
```

Torna o uso da função inviável, pois a operação de potência é cara, consequentemente, torna o processamento da função de hash lento e, desta forma, atrapalha completamente o tempo de operação das operações da estrutura de tabela hash. Mas há uma forma de resolver isto com o uso do método de Horner [3] para a avaliação do polinômio. Com este método, conseguimos avaliar o polinômio sem o uso da função de potência, apenas reescrevendo este polinômio. Neste caso, a implementação ficaria sendo:

```
1  public int hash(String s, int M)
2  {
3      int p = 31;
4      int sum = s.charAt(0);
5
6      for (int i = 1; i < s.length(); i++)
7      {
8          sum = sum * p + s.charAt(i);
9      }
10
11     return (sum % M);
12 }
```

[2] Calculation of the hash of a string - https://cp-algorithms.com/string/string-hashing.html
[3] Horner's method - https://en.wikipedia.org/wiki/Horner%27s_method

Sabendo disto, a tabela hash implementada neste trabalho faz uso desta função de hash e faz uso do método de Horner para a avaliação do polinômio da função. Além disso, como a função pode retornar valores negativos e queremos que o valor retornado seja um índice na tabela, é feito um "Math.abs()" no resultado final para que seja retornado um valor positivo. A implementação da função de hash como aqui descrita está no método "getIndex()" da classe *HashTable*.

A estrutura de árvore AVL implementada faz uso de um nó que armazena seus filhos esquerdo e direito, a altura, a chave e um *IndexEntry*. Como a árvore AVL é uma estrutura de árvore balanceada, foram necessárias as implementações das operações de balanceamento da árvore que, neste caso, são as operações de rotação. As operações de rotação podem ser a esquerda ou a direita, sendo estas rotações implementadas nos métodos "rightRotation()" e "leftRotation()" da classe *dev.ltoscano.indexer.structure.AVL.AVL*. Além disso, a operação de balanceamento está no método "rebalance()" da mesma classe.

A cada inserção na estrutura, os fatores de balanceamento dos nós são verificados, ou seja, é verificado se algum nó fere a propriedade de balanceamento da árvore AVL, onde um nó é considerado desbalanceado se a altura de suas subárvores diferem em mais de 1 unidade. Caso o nó esteja desbalanceado, é necessário a aplicação de rotações simples ou duplas para a esquerda ou direita. A decisão de qual o tipo de rotação é necessária, depende da altura das subárvores dos nós. As imagens a seguir[4] exemplificam a realização destas rotações simples e duplas para cada caso de nó desbalanceado:

Figura 1 – Rotação simples esquerda

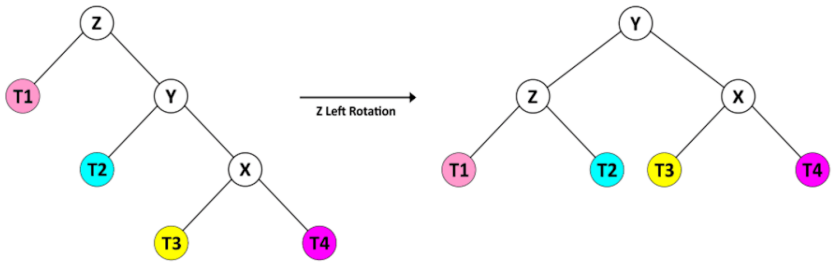


Figura 2 – Rotação simples direita

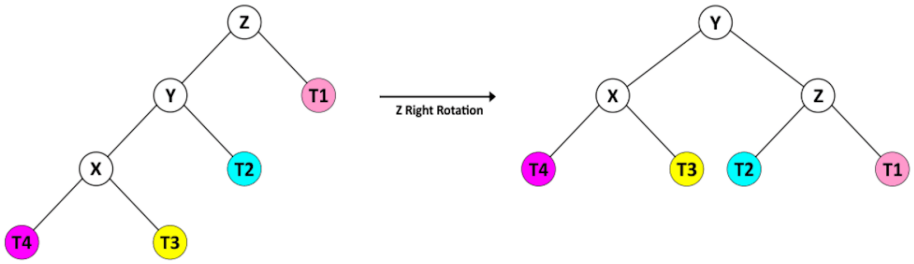
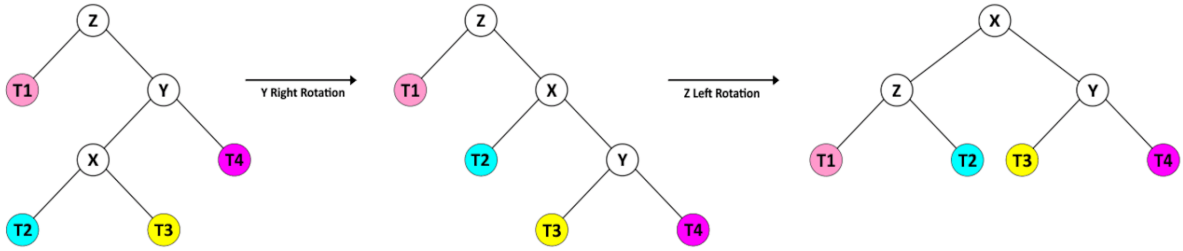
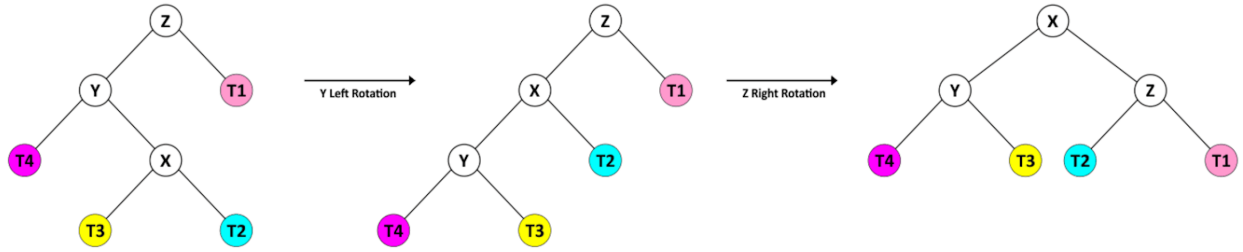


Figura 3 – Rotação dupla direita/esquerda



[4] Guide to AVL Trees in Java - https://www.baeldung.com/java-avl-trees

Figura 4 – Rotação dupla esquerda/direita



Por fim, a estrutura de dados do tipo Trie R-way implementada faz uso de um nó que além das chaves e de um objeto *IndexEntry*, possui um vetor de filhos de tamanho 36, pois cada nó filho é referente a um tipo de caractere possível nas palavras que o sistema aceita, ou seja, letras minúsculas de "a" até "z" e os dígitos "0" a "9".

Diferente das outras estruturas de dados em que a chave (Palavra) é indivisível, nesta estrutura cada nó da árvore é um caractere de uma chave, ou seja, existe um caminho a ser percorrido na árvore de acordo com os caracteres de uma chave. O nó que identifica que uma chave foi encontrada na árvore é caracterizado pela presença de um objeto *IndexEntry* neste nó. Desta forma, os nós intermediários não armazenam o objeto *IndexEntry*, sendo este presente apenas no último nó que é referente ao último caractere da chave.

Ainda em relação a Trie implementada, o índice dos filhos são referentes aos códigos dos caracteres na tabela ASCII. Em outras palavras, no vetor de filhos de tamanho 36, as primeiras posições são reservadas para as letras minúsculas, que na tabela ASCII começam no código decimal 97 e terminam no código 122. Desta forma, ocorre um deslocamento do tipo "$(charCode - 97)$". Assim, a letra "a" é o filho da posição 0 do vetor e a letra "z" é o filho da posição 25. A partir daí, temos os dígitos de 0 a 9, que na tabela ASCII começam no código 48 e terminam no código 57, neste caso, o deslocamento aplicado será do tipo "$(charCode - 22)$" pois, desta forma, o dígito "0" começará na posição 26 do vetor e o dígito "9" terminará na posição 35. Também, é importante destacar que apesar do vetor ter tamanho 36, suas posições só são alocadas conforme a necessidade, ou seja, quando um nó para aquela posição deve ser criado. Isto ajuda a reduzir o consumo de memória da estrutura. O mesmo também acontece na tabela hash implementada, onde somente as posições utilizadas que são alocadas, sendo nulas todas as outras.

2.3 Outros detalhes sobre o sistema

Outros pontos menos importantes do trabalho são em relação aos modos de execução presentes no sistema, sendo eles: Web, Console e Teste.

No modo Web, o sistema é inicializado em seu modo padrão, ou seja, uma aplicação web. Neste modo, o servidor *Apache Tomcat* será inicializado para servir as páginas web ao usuário. O funcionamento deste modo se baseia em algumas páginas, sendo a página principal uma tela de carregamento. Quando o usuário acessa a página inicial do sistema, a tela de carregamento será retornada, dando início a construção do índice de notícias, caso ele ainda não tenha sido criado. Após isto, o usuário é redirecionado automaticamente para uma página de busca, onde pode inserir seus termos de busca. Apesar do sistema ter como requisito permitir que o usuário busque por até dois termos, a implementação que foi feita não faz esta restrição, permitindo que o usuário busque por uma lista de termos. Digitando os termos na caixa de busca e acionando o botão de busca, o sistema redirecionará o usuário para uma nova página que será a página de resultados, que contém uma lista com os 20 resultados da busca ordenados pela sua relevância, sendo estes resultados as notícias, que podem ser acessadas diretamente ao clicar no título da notícia. As figuras **??**, **??**, **??** e **??** mostram estas páginas. O modo console, permite acesso pelo usuário às mesmas funcionalidades, porém, sem interface gráfica. Desta forma, o usuário interage via console diretamente com o índice de notícias, executando as operações de construção do índice, realização de consultas e obtenção de estatísticas. Por

fim, o modo teste executa todas os experimentos realizados neste trabalho, criando como resultado final arquivos CSV que são armazenados na pasta "Results"do projeto (Este não é um modo de interação com o usuário).

Por fim, para a execução do projeto em cada um destes modos, existe um script *"run"* que faz a compilação e execução do projeto. Este script repassa os parâmetros para a aplicação que então configura seus parâmetros de inicialização. As configurações da aplicação podem ser vistas na classe "*dev.ltoscano.indexer.configuration.AppCnfig*". Nem todas as propriedades estão parametrizadas, porém, a maioria está. Por exemplo, é possível alterar o modo de execução da aplicação, o caminho do dataset de notícias, a estrutura de dados que será utilizada para o índice, o limite de resultados da consulta, os arquivos de consultas para os testes, a quantidade de execução dos testes. Alguns exemplos de parâmetros que podem ser utilizados são:

- <modo> = 0, 1, 2 (Sendo eles: Web = 0, Console = 1, Teste = 2)
- <datasetPath> = Caminho absoluto do arquivo contendo o dataset de notícias
- <indexStructure> = 0, 1, 2 (Sendo eles: HashTable = 0, AVL = 1, Trie = 2)
- <queryLimit> = Limite de resultados da consulta
- <randomOneWordTestPath> = Caminho do arquivo aleatório com consultas de 1 palavra
- <randomTwoWordTestPath> = Caminho do arquivo aleatório com consultas de 2 palavra
- <bestOneWordTestPath> = Caminho do arquivo com consultas de 1 palavra com sucesso
- <avgOneWordTestPath> = Caminho do arquivo com consultas de 1 palavra com metade de sucessos e a outra metade de fracassos
- <worstOneWordTestPath> = Caminho do arquivo com consultas de 1 palavra somente com fracassos
- <bestTwoWordTestPath> = Caminho do arquivo com consultas de 2 palavra com sucesso
- <avgTwoWordTestPath> = Caminho do arquivo com consultas de 2 palavra com metade de sucessos e a outra metade de fracassos
- <worstTwoWordTestPath> = Caminho do arquivo com consultas de 2 palavra somente com fracassos

Estes parâmetros podem ser usados em sistemas Unix como:

- ./run <modo> <datasetPath> <indexStructure> <queryLimit>, para o modo Web ou Console
- ./run <randomOneWordTestPath> <randomTwoWordTestPath> <bestOneWordTestPath> <avgOneWordTestPath> <worstOneWordTestPath> <bestTwoWordTestPath> <avgTwoWordTestPath> <worstTwoWordTestPath>, para o modo Teste

Caso ainda reste dúvidas, olhe o método "parseArgs()" das classes *IndexerWebApplication*, *IndexerConsoleApplication* e *IndexerTestApplication* do pacote *dev.ltoscano.indexer*.

3 Experimentos

Os experimentos realizados neste trabalho consistiram na obtenção do consumo de tempo e memória para determinadas operações: Tempo de construção do índice, tempo de inserção no índice, consumo de memória do índice criado, tempo de consulta para 1 ou 2 palavras aleatórias, tempo de consulta para melhor caso, caso médio e pior caso de consultas com 1 ou 2 palavras, tempo de busca no índice, tempo do cálculo de relevância e tempo de ordenação do resultado.

Foram considerados como melhor caso, caso médio e pior caso das consultas, os casos em que todas as consultas possuem termos que estão no índice, os casos em que metade das consultas possuem termos que estão no índice e a outra metade não e, por fim, os casos em que todas as consultas possuem termos que não estão no índice, respectivamente.

As consultas em que todos os termos estão no índice, foram obtidas das palavras que estão no próprio índice construído sobre o dataset de notícias utilizado neste trabalho. As consultas em que todos os termos não estão no índice, foram obtidas a partir de geradores "*Lorem ipsum*" [5], que por gerarem palavras em latim, provavelmente, não estarão no índice de palavras em inglês. Por fim, o "caso médio" se constuitui de metade das palavras do melhor caso e metade do pior caso, assim, metade das consultas terá sucesso e a outra metade não. As consultas aleatórias, consistem de arquivos gerados com uma proporção aleatória de termos que estão no índice e de termos que não estão no índice. Em todos os casos, são 10 mil consultas que serão realizadas sobre o índice de notícias. Estes arquivos gerados se encontram no diretório "*src/main/resources/dataset*" do projeto.

A coleta de dados foi realizada em um computador com AMD FX 8300, 16 GBs de RAM com Ubuntu 18.04. Além disso, destaco que os tempos obtidos utilizando o OpenJDK 8 foram piores que os tempos obtidos utilizando o Oracle JDK 8, desta forma, os tempos podem ser diferentes dos obtidos em outra configuração de hardware e/ou software. Para este trabalho, o Oracle JDK 8 foi o utilizado.

Para a obtenção de médias de tempo e consumo de memória, foram executados 25 rodadas de teste para cada estrutura de índice do trabalho. Cada rodada de teste consiste na criação de um índice e realização de consultas sobre ele, sendo estas consultas obtidas dos arquivos gerados das formas já descritas.

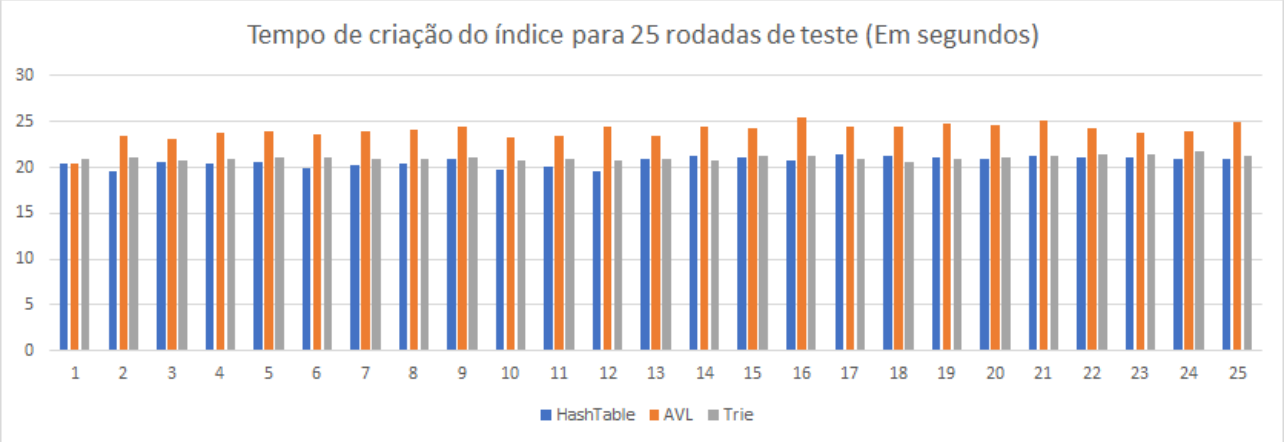


Figura 5 – O tempo de criação do índice consiste no tempo de execução do método "buildIndex()" da classe *dev.ltoscano.indexer.index.Index*, ou seja, é o tempo de inserção das entradas de índice na estrutura de dados utilizada e o cálculo dos pesos IDF dos documentos.

[5] https://www.lipsum.com/

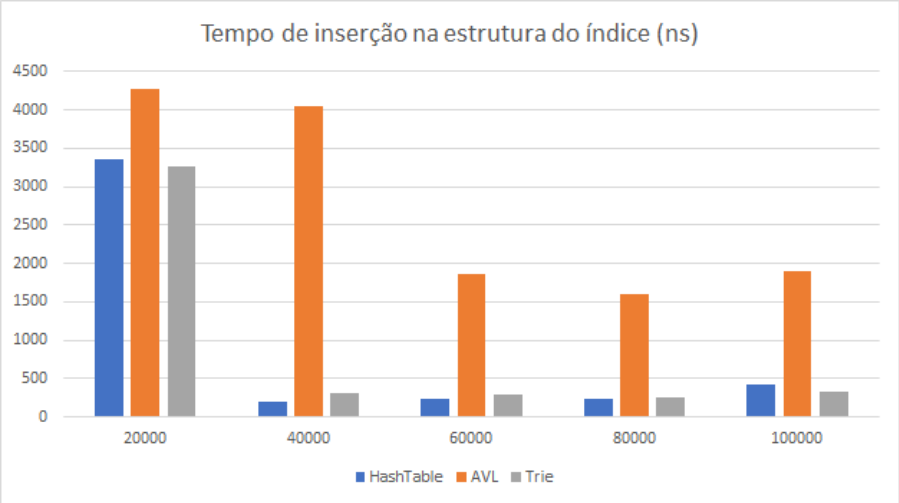


Figura 6 – O tempo de inserção no índice é o tempo gasto para inserir uma entrada no índice, quando este índice já está com uma determinada quantidade de entradas. Neste caso, o gráfico mostra resultados médios para inserção quando o índice possui 20 mil, 40 mil, 60 mil, 80 mil e 100 mil entradas.

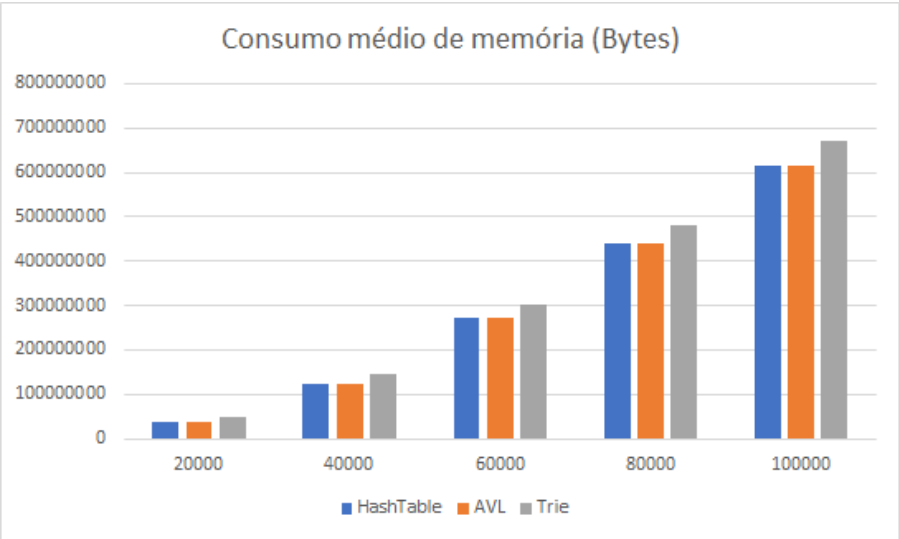


Figura 7 – O consumo de memória é o consumo total de memória pela aplicação após uma determinada quantidade de inserções na estrutura de índice de notícias. Neste caso, o gráfico mostra resultados médios do consumo de memória quando o índice possui 20 mil, 40 mil, 60 mil, 80 mil e 100 mil entradas. É importante destacar que os dados são aproximados, pois a linguagem Java possui memória gerenciada pela JVM, o que dificulta o gerenciamento de memória por parte do programador.

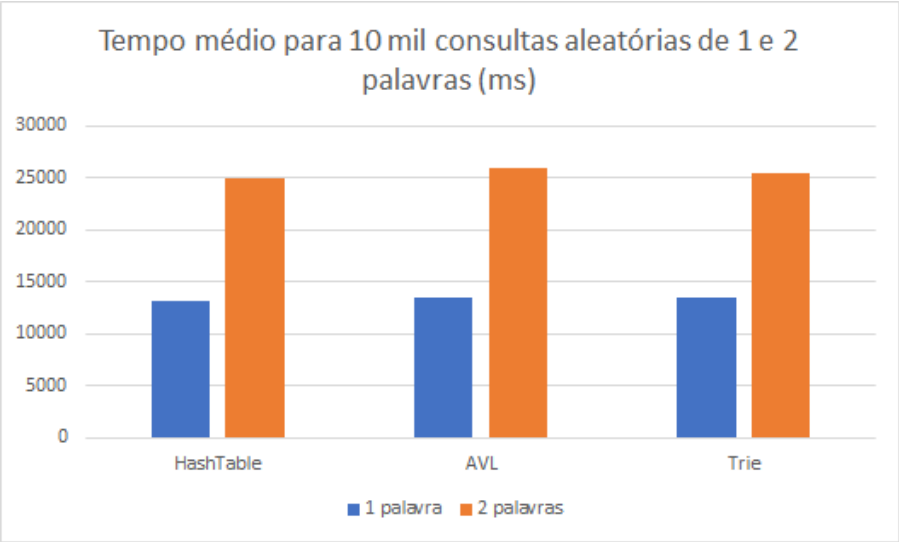


Figura 8 – O gráfico mostra o tempo total médio, em milisegundos, para a realização de 10 mil consultas de 1 e 2 palavras aleatórias

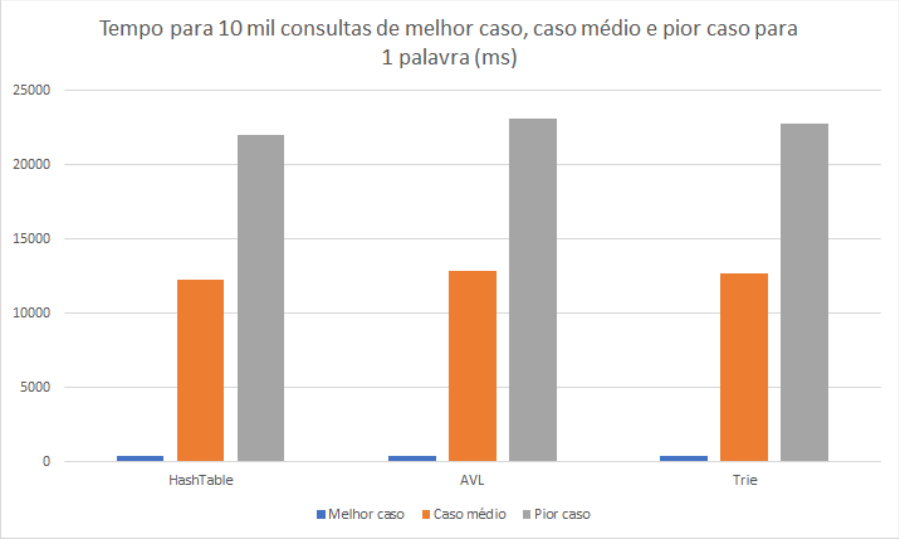


Figura 9 – O gráfico mostra o tempo total médio, em milisegundos, para a realização de 10 mil consultas de melhor caso, caso médio e pior caso com 1 palavra.

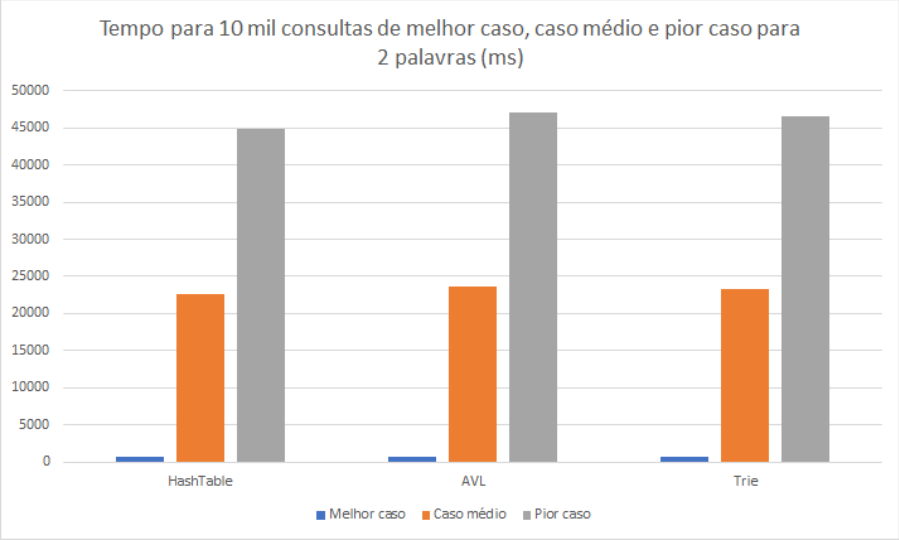


Figura 10 – O gráfico mostra o tempo total médio, em milisegundos, para a realização de 10 mil consultas de melhor caso, caso médio e pior caso com 2 palavras.

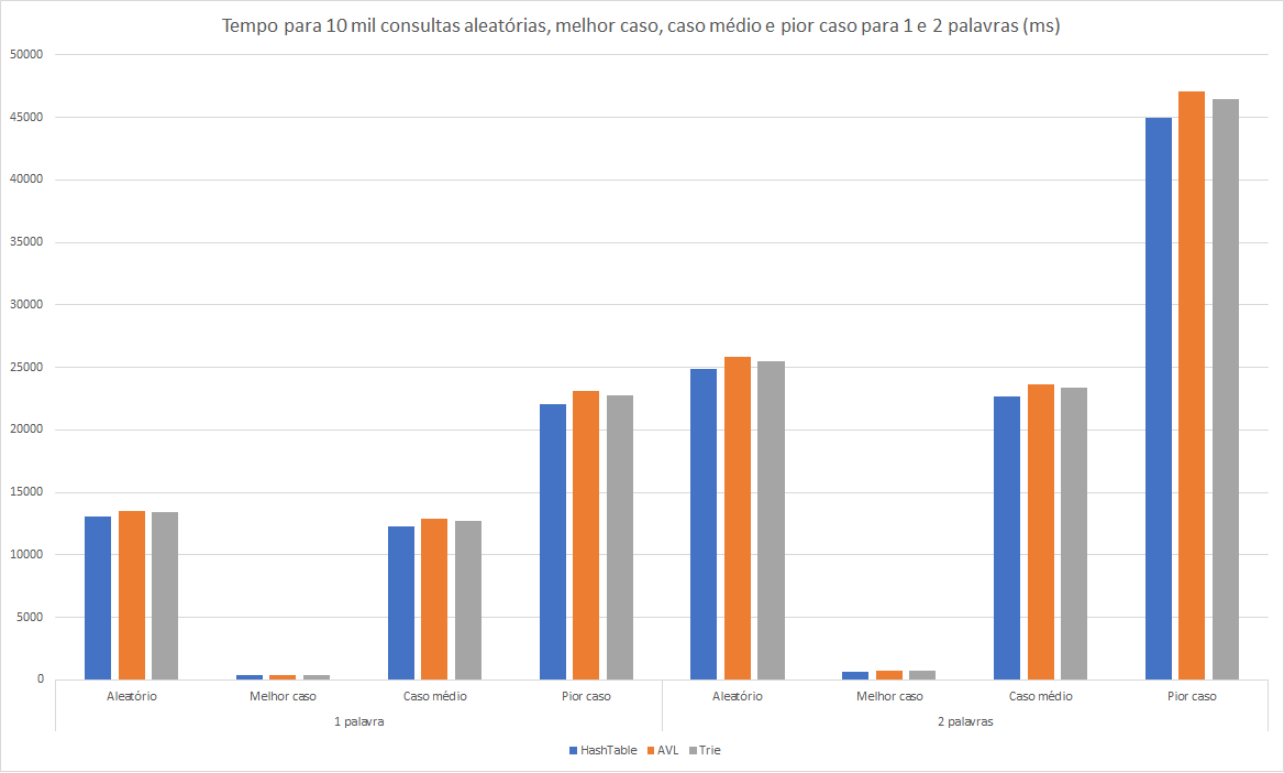


Figura 11 – Gráfico comparativo com todos os casos de consultas testados

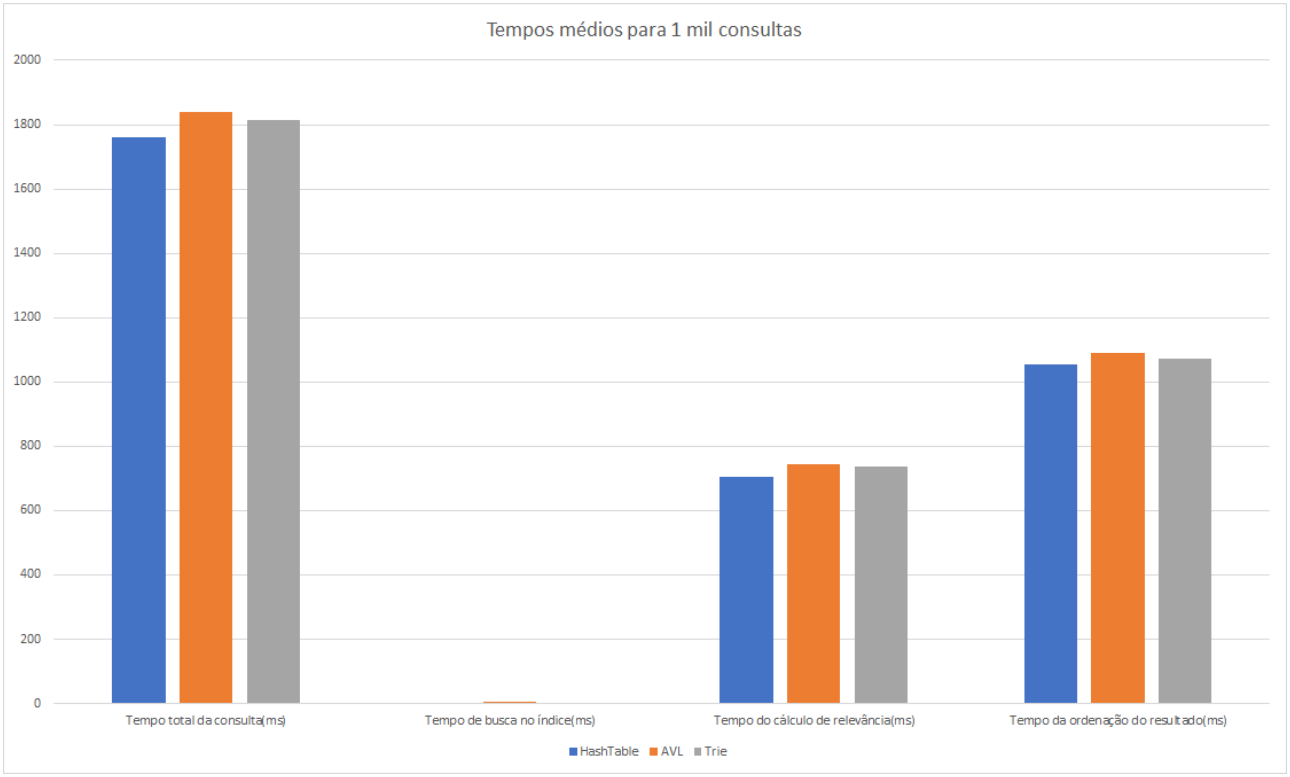


Figura 12 – Gráfico com os tempos médios para mil consultas, sendo que o tempo total para uma consulta é o tempo para obtenção da entrada de índice a partir da estrutura do índice, cálculo da relevância dos documentos e ordenação dos resultados.

4 Considerações

Sobre os resultados obtidos neste trabalho é possível destacar que em relação ao tempo de construção dos índices, a árvore AVL teve os piores tempos, tendo uma média de 24 segundos com mínimo de 20 segundos e o máximo de 25 segundos. Já em relação às estruturas de tabela hash e trie, ambas obtiveram tempos de construção muito semelhantes, com uma média de 20 segundos, mínimo de 19 segundos e máximo de 21 segundos para a tabela hash e uma média de 21 segundos, mínimo de 20 segundos e máximo de 21 segundos para a trie, ou seja, a trie teve seu tempo de construção levemente superior ao da tabela hash, sendo esta estrutura a de menor tempo de construção neste trabalho.

Ainda em relação ao tempo de construção, é importante ser destacado que é o tempo envolvido na construção de todo o índice, incluindo o pré-cálculo dos pesos dos termos nos documentos. O objetivo desta implementação foi o de gastar mais tempo na construção da estrutura, que seria uma operação única durante a execução do sistema, para ganhar mais tempo depois nas operações mais frequências que seriam as consultas sobre o índice de notícias.

Em relação ao tempo de inserção na estrutura do índice conforme o índice cresce. Novamente, é possível perceber que a árvore AVL se mostrou bem mais lenta que as demais estruturas, provavelmente, devido às operações de balanceamento que a árvore executa nas inserções. Destaco que nos tempos de inserção com 20 mil entradas no índice, todas as estruturas possuem um tempo alto de inserção, pois a primeira inserção na estrutura de dados gerou tempos extremamente discrepantes em relação às demais, possivelmente, devido a algum processo interno da JVM que faz com que na primeira alocação da estrutura, um grande tempo seja consumido. Por causa destes tempos discrepantes, a média é afetada para todas as estruturas nesta faixa de 20 mil entradas. Para as demais, o problema parece não ocorrer. Desta forma, olhando somente para a inserção na faixa de 40 a 100 mil entradas de índice, é possível perceber que a AVL é bastante custosa em relação às demais quando há menos dados na estrutura, diminuindo este custo com o aumento da estrutura. É possível que isto seja causado por uma grande quantidade de rotações enquanto a estrutura está em sua fase inicial e isto ocorra em menor quantidade para uma estrutura de árvore maior. Outro ponto de destaque acontece na faixa de 100 mil entradas de índice, onde o tempo de inserção na tabela hash se inverte com a trie, enquanto abaixo disto, o tempo de inserção é menor na tabela hash do que na trie. Provavelmente, isto ocorre pelo aumento no número de colisões na tabela hash, já que neste ponto, há uma grande quantidade de elementos na tabela, aumentando a chance de colisão.

Em relação ao consumo de memória da aplicação conforme o índice de notícias cresce, não podemos afirmar muitas coisas, porque a medição neste caso não foi precisa, já que a linguagem Java é uma linguagem que possui gerenciamento de memória pelo *Garbage Collector* da JVM. Mas ainda assim, é possível perceber um consumo maior da aplicação quando o uso de uma Trie é feito. Se este padrão foi real, é possível explicá-lo pelo tipo de implementação feito, que seria o da Trie R-way, onde os filhos de um nó são a quantidade de símbolos possíveis no alfabeto que, neste caso, são 36 símbolos, ou seja, cada nó da árvore pode ter 36 filhos. Desta forma, a Trie R-way tem um potencial bem maior de consumo de memória RAM do que as demais estruturas. Comparando o gasto de memória da tabela hash com a AVL, elas se mostraram com um consumo muito próximo, como pode ser visto na figura 7.

Em relação aos tempos de consulta no índice para 1 e 2 palavras, vários casos foram testados. Tomando como exemplo a figura 11 que mostra um comparativo de todos estes casos de consultas, podemos perceber as relações de complexidade de cada caso das estruturas de dados. No caso das consultas que foram geradas para “o melhor caso”, que são as consultas de 1 ou 2 termos que estão na estrutura de índice, estamos com a situação em que todas as consultas terão sucesso. Já no “pior caso”, são as consultas de 1 ou 2 termos em que as palavras não estão na estrutura do índice, desta forma, todas as consultas não terão sucesso. Assim, estamos perdendo sempre uma grande quantidade de tempo, pois estamos executando a busca até o “final” das estruturas. Por isso, a grande diferença de tempo entre o melhor caso e o pior caso, sendo o primeiro um caso de menor esforço e o segundo de maior esforço. O “caso médio”, seriam as consultas com 1 ou 2 palavras contendo metade das palavrás que estão no índice e a outra metade não, desta forma, metade das consultas terá sucesso e a outra metade falhará. Com o gráfico, é possível perceber claramente isto, com este caso se situando

na metade do tempo entre as consultas de melhor e pior caso. Por fim, temos os casos aleatórios, em que temos consultas de 1 ou 2 termos com uma proporção aleatórias de termos que estão ou não no índice. Neste caso, é possível perceber que o tempo destas consultas está situado próximo das consultas de caso médio. Assim, podemos afirmar que no uso real do sistema, os tempos de consulta estariam próximos do caso médio. Para 10 mil consultas, isto seria um tempo próximo dos 12 milisegundos para 1 um termo e 23 milisegundos para 2 termos.

Ainda sobre estes tempos, temos uma leve vantagem para tabela hash, seguido pela trie e, por fim, pela árvore AVL. A explicação para estes tempos tão próximos é, provavelmente, devido à implementação que foi feita neste trabalho. Podemos ver na figura 12 o tempo total para mil consultas e quanto deste tempo é usado para buscar a entrada no índice em si, o tempo para o cálculo de relevância dos documentos e o tempo de ordenação dos resultados. Da forma em que este trabalho está implementado, o tempo de busca de uma entrada do índice é irrelevante em relação aos tempos de cálculo de relevância e ordenação dos resultados, sendo o tempo gasto para a ordenação dos resultados o maior deles.

Em outras palavras, para cada consulta realizada no sistema, passamos mais tempo ordenando os resultados da consulta do que buscando as entradas no índice. A implementação do cálculo de relevância também gasta um tempo considerável do tempo total de uma consulta. Desta forma, caso no futuro quisermos melhorar o tempo das consultas para este sistema, devemos melhorar a implementação do cálculo de relevância e do algoritmo de ordenação dos resultados.

5 Extra

Na quinta-feira (10/10) deletei por erro meu trabalho inteiro e que estava praticamente pronto, faltando apenas terminar a implementação da estrutura de dados SkipList e coletar os dados de teste relacionados. Eu possuía backups, porém, estavam desatualizados há 4 dias e, desta forma, acabei perdendo uma parte considerável deste trabalho, tendo que refazê-las. Acabei não conseguindo terminar a implementação da SkipList a tempo da entrega, pois antes mesmo de ter perdido esta parte do trabalho, estava tendo alguns problemas em relação a esta estrutura, com a minha implementação demorando por volta de 30 minutos apenas para a construção do índice de notícias. Desta forma, não sei se estes tempos que eu estava obtendo com esta estrutura seriam os esperados ou se minha implementação que estava muito ruim, mas se este for o esperado para esta estrutura no cenário deste trabalho, ela seria de longe a pior estrutura para ser utilizada neste índice.