FeiraGugou – Motor de busca de palavras-chave em textos

Johnny Quest Dantas Pereira

Curso de Graduação em Engenharia de Computação – Universidade Estadual de Feira de Santana (UEFS). Feira de Santana, BA – Brasil.

johnny.quest.dp@gmail.com

# 1. Introdução

O serviço de busca do Google foi um dos pioneiros nesse ramo, surgindo em uma época onde não existia uma ferramenta muito eficiente para fazer buscas na internet a partir de palavras-chave. Devido a isso, a popularidade do Google Search foi explosiva, visto que até hoje é o principal mecanismo de filtragem de páginas web do mundo. Seu funcionamento é rápido e amigável, basta digitar um conjunto de palavras e o mecanismo irá exibir todas as páginas que possam ser de interesse do pesquisador, sendo aplicados diversos algoritmos para determinar quais páginas são potencialmente relevantes, para que essas sejam as primeiras a serem exibidas ao usuário.

Inspirado na grande importância para a humanidade que esse serviço possui, foi proposto aos estudantes do segundo semestre de Engenharia de Computação da UEFS que desenvolvessem um sistema de filtragem de conteúdo inspirado no Google, que deveria pesquisar em um conjunto de arquivos texto armazenados na memória da máquina, uma palavra inserida pelo usuário, e ordenar os resultados de acordo com a sua “relevância”, sendo que essa relevância seria a quantidade de vezes que tal palavra aparece em tal arquivo. O usuário pode então acessar qualquer uma das páginas resultantes de sua pesquisa dentro do programa.

A partir da necessidade de ser um sistema eficiente, um dos requisitos da proposta é que o programa use uma estrutura de dados que permita que operações de busca, inserção e remoção de informações tenham complexidade O(log n), ou seja, que não seja necessário verificar todo o conjunto de dados para encontrar um específico, e sim “dividir para conquistar” (divide and conquer), separando o conjunto em vários subconjuntos tal que o programa possa procurar um dado apenas nos subconjuntos onde ele poderia estar.

Além da busca, o usuário poderá ter acesso a uma lista com as top palavras mais buscadas e as top páginas mais acessadas, tanto essas duas listas quanto os resultados de uma busca devem poder ser ordenados de duas formas: do mais relevante ao menos relevante e vice-versa, à escolha do usuário.

Por fim, o programa deve acompanhar um repositório de “páginas web” (que na verdade são arquivos .txt) onde existam 15 palavras tal que cada uma delas apareçam em pelo menos 5 páginas.

**2. Fundamentação Teórica**

**2.1. Árvore AVL**

Essa estrutura de dados consiste em uma árvore binária de pesquisa onde, para cada nó, a sub-árvore à esquerda dele contenha valores menores que ele, e a da direita contenha valores maiores. O que diferencia a árvore AVL de outras árvores de pesquisa é a propriedade da altura/balanceamento, caracterizada como “para cada nó interno v de T, as alturas dos filhos de v podem variar em no máximo 1” (GOODRICH & TAMASSIA, 2004), ou seja, sempre que as alturas dos filhos de um nó variarem em valores maiores que 1, será feito um balanceamento para manter a árvore nos padrões de uma AVL, consequentemente transformando todas as suas sub-árvores em árvores AVL também. Os processos de inserir valores, remover e buscar em uma árvore AVL possuem complexidade O(log n).

**2.2. Serialização**

O processo de serialização baseia-se em salvar instâncias de objetos no armazenamento do computador, no estado em que eles se encontram, tendo a possibilidade de “recuperá-los” durante a execução do programa. “O processo de serialização de objetos permite converter a representação de um objeto em memória para uma sequência de bytes(...)” (RICARTE, 2001), essa sequência de bytes pode ser lida através da desserialização, que seria o processo inverso, onde o arquivo é “convertido” para um objeto do programa. É importante notar que as classes cujos objetos serão salvos devem implementar a interface Serializable.

**2.3. Expressões regulares**

São formas de identificar uma cadeia específica de caracteres que tem diversas aplicabilidades, e para o programa em questão, foi utilizada para remover vírgulas, pontos, parênteses e afins dos textos “adquiridos” dos arquivos, tendo assim apenas as palavras (conjunto de caracteres alfanuméricos) válidas para o programa, esse processo “(...) envolve, como entrada, uma string de assunto, uma expressão regular e uma string de substituição. A saída é a string de assunto com todas as correspondências da expressão regular trocadas pelo texto de substituição.” (GOYVAERTS & LEVITHAN, 2009), ou seja, foi utilizado um método que “pega” apenas os caracteres desejados de uma string, esses caracteres desejados são informados através de uma expressão regular.

**2.4. Tratamento de exceções**

Existem vários tipos de erros que podem ocorrer em tempo de execução do programa, os chamados Exceptions. Caso algum método tenha a chance de dar algum erro (se ele “dá” throws em alguma Exception), essa possibilidade deve ser tratada através de uma rotina try/catch. Esta rotina consiste em dois blocos de código, tal que “se algum código dentro do bloco try lançar uma exceção da classe especificada na cláusula catch, então, o programa pula o restante do código no bloco try e executa o código de tratamento dentro da cláusula catch. ” (HORSTMANN & CORNELL, 2009).

**2.5 Busca Binária**

É um processo de busca por um dado em um conjunto ordenado que se baseia em “dividir para conquistar”. Ele divide a região de busca pela metade a cada iteração, comparando o dado procurado com o valor “do meio” para saber se aquele objeto se encontra “à esquerda” ou “à direita” do valor determinado como valor do “meio”.

**2.6 Algoritmo de Ordenação Quick-Sort**

Esse é um dos algoritmos de ordenação mais eficientes que existe, ele também é baseado na premissa de “dividir para conquistar”, onde o algoritmo irá selecionar um valor do conjunto para ser um “pivô”, e irá separar esse conjunto em três: os valores menores que o pivô, os valores maiores que o pivô, e o pivô, que automaticamente ficará na posição correta (entre os outros dois conjuntos), então, esses dois conjuntos continuarão sendo divididos até terem tamanho 1, em seguida que eles voltam, de forma recursiva, sendo concatenados um por um, até resultar em um único conjunto ordenado.

**3. Metodologia**

A solução para o problema foi alcançada através de diversas sessões de discussão utilizando a metodologia Problem-Based-Learning(PBL), onde os 9 integrantes debatem sobre como resolver cada problema que venham a se deparar durante o desenvolvimento do programa.

A primeira dúvida apresentada foi como deveríamos ler as palavras dos arquivos. Tal situação foi resolvida apresentando métodos das classes BufferedReader, mais especificamente o readLine(), que retorna uma String correspondente a todos os caracteres presentes em uma linha do arquivo. Porém, era necessário armazenar palavra por palavra, então foi apresentado que poderíamos separar uma String em um conjunto de palavras utilizando o método split() da classe String.

Isso estava funcionando, porém, as palavras estavam sendo salvas com caracteres indesejados, como vírgulas, pontos e afins. Esse problema foi resolvido utilizando uma expressão regular (regex) dentro de um replaceAll, dessa forma poderia-se determinar quais caracteres deveriam ser aproveitados e quais deveriam ser descartados.

Para a estrutura de dados que faça operações de inserção, remoção e busca com complexidade O(log n), foi proposto utilizar uma Árvore AVL. No código em questão, foi implementado um algoritmo adaptado de Rodrigo Vilar, disponível em: <https://gist.github.com/rodrigovilar/5754425#file-arvoreavl-java>. (Último acesso em 27/06/2018).

Para tratar as possíveis modificações dos arquivos texto, foi proposto armazenar os horários de modificação de cada um dos arquivos e, sempre que for lê-lo novamente, comparar a sua data de modificação com a armazenada. Caso sejam diferentes, significa que o arquivo foi modificado depois de a árvore ter sido gerada, então o tratamento poderia ser feito a partir daí. Porém, no código em questão, foi decidido que a cada pesquisa, o programa irá reler os arquivos para atualizar a quantidade de ocorrências de cada uma das palavras, e adicionar novas palavras caso existam.

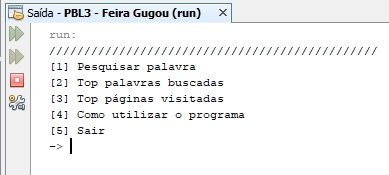
A estrutura geral do programa em questão foi decidida da seguinte forma: há uma árvore principal de palavras, cada palavra possui uma árvore de arquivos em que ela aparece, cada um armazenando também a sua quantidade. Esses arquivos são referências à arquivos que se encontram armazenados em uma segunda árvore à parte. Dessa forma, foi possível gerar o top-K palavras mais buscadas, pegando todas as palavras da árvore principal e ordenando em relação à quantidade de pesquisas, e o top-K arquivos mais buscados, onde os arquivos da segunda árvore são ordenados em relação à quantidade de acessos.

O repositório de testes do respectivo programa, que é um dos requisitos do problema, é composto por 20 arquivos com trechos da Bíblia, que são textos extensos e com muitas palavras diferentes, o que é excelente para testar a eficiência do sistema. Certamente haverão várias palavras que se repetem em pelo menos 5 arquivos, entretanto, as seguintes podem ser usadas como parâmetro: “Deus”, “mas”, “anjos”, “províncias”, “terra”, “mar”, “senhor”, “judeus”, “então”, “rei”, “noite”, “rainha”, “dia”, “trono”, “vida”.

O programa foi desenvolvido em linguagem Java, utilizando a Interface de Desenvolvimento Integrado (IDE) Netbeans versão 8.2, com o Java Development Kit (JDK) 8u111, no sistema operacional Windows 10 Pro 64 bits e testado em um computador com processador Intel Core i3-2120 e 8gb de memória DDR3 e no mesmo não foi notado nenhuma demora para ler os arquivos.

# 4. Resultados e Discussões

Ao executar o programa, o usuário se depara com o menu principal do programa, que é uma interface de linhas de comando que lhes dá uma série de opções, cada uma com um número correspondente, e aguarda a entrada de uma resposta do usuário, como demonstrado na Figura 1.



**Figura 1 – Menu principal do programa (Fonte: próprio autor)**

Ao escolher pesquisar uma palavra, o programa irá ler todos os arquivos .txt que estejam na pasta “repositório” (dentro do diretório do programa), e armazenar todas as palavras encontradas em uma Árvore Binária balanceada (AVL). Cada palavra terá uma referência pra uma lista de arquivos onde ela aparece, e quantas vezes aparece em cada arquivo (Mais especificamente, na verdade as palavras possuem uma árvore de “Aparições”, e a cada aparição possui a quantidade de vezes que aquela palavra se encontra naquele arquivo, e a referência do arquivo em si, sendo que esse arquivo é um objeto Arquivo, que se encontra em uma segunda árvore de arquivos, e possui a referência de um arquivo e um atributo que representa a quantidade de vezes que o mesmo foi acessado.)

Assim que essa árvore de palavras é gerada, o programa irá realizar nela uma busca pela palavra inserida pelo usuário. Caso a encontre, será impresso na tela todas as aparições dessa palavra, ordenadas de acordo com a sua relevância (quantidade de vezes que a palavra aparece no arquivo) em ordem crescente ou decrescente, à escolha do usuário. Caso não encontre, essa palavra será adicionada na árvore, e em ambos os casos a quantidade de vezes que essa palavra foi pesquisada é somada em 1.

Se o usuário pesquisar uma palavra que esteja presente em algum dos arquivos, ele pode escolher qualquer uma das aparições resultadas para exibir o seu conteúdo na tela, como mostrado na Figura 2.

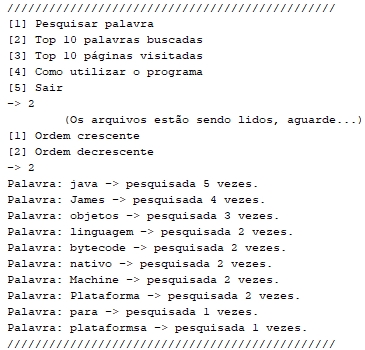
# 

# Figura 2 – Pesquisa de uma palavra e acesso a um arquivo. (Fonte: próprio autor)

Resumindo: o programa leu todos os arquivos da pasta repositório, adicionou todas as palavras encontradas com suas respectivas aparições em uma árvore, e todos os arquivos com suas respectivas quantidades de acessos em uma segunda árvore. Em seguida o usuário digitou a palavra “java”, o programa buscou “java” na árvore e imprimiu todas as aparições que estavam presentes nesse objeto, cada uma com um número representante (na ordem determinada). O usuário decidiu acessar o arquivo c.txt, representado pelo número 1, então o programa imprimiu todo o conteúdo desse arquivo na tela.

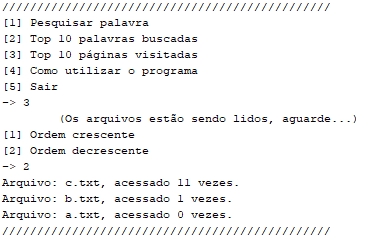
Como todo objeto Palavra possui um atributo que representa a quantidade de vezes que ela foi pesquisada e todo objeto Arquivo possui um atributo que representa a quantidade de vezes que ele foi acessado, é possível ordená-los a partir dessas características, possibilitando assim gerar uma lista com as “top” palavras, e fazer o mesmo com os arquivos. É exatamente o que acontece quando o usuário seleciona a opção 2 ou 3 do menu principal.

Caso o usuário entre na opção 2 do menu(top palavras buscadas), o programa irá gerar uma lista de palavras a partir da árvore principal, e ordenar essas palavras de acordo com a quantidade de buscas de cada uma (de forma crescente ou decrescente, à escolha do usuário), como demonstrado na Figura 3.



**Figura 3 – Top palavras buscadas (Fonte: próprio autor)**

E, se escolher a opção 3 do menu (top arquivos acessados), o programa irá gerar uma lista de arquivos a partir da segunda árvore (árvore de Arquivos), ordenados a partir de suas respectivas quantidades de acessos (o usuário também pode escolher entre crescente e decrescente), como demonstrado na Figura 4.



**Figura 4 – Top páginas acessadas (Fonte: próprio autor)**

Para garantir que tudo esteja funcionando adequadamente, foi realizada a seguinte rotina de testes (utilizando o repositório que foi entregue junto ao programa):

As seguintes palavras foram pesquisadas: (Em todas elas, os arquivos resultados foram analisados manualmente um a um, para garantir que os resultados exibidos estavam corretos.)

Deus -> 2 vezes Amor -> 3 vezes

Senhor -> 4 vezes Cristo -> 5 vezes

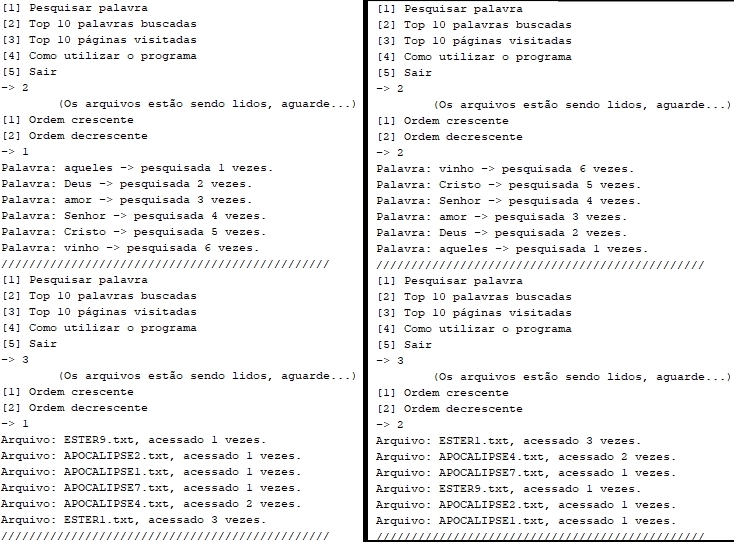
Aqueles -> 1 vez Vinho -> 6 vezes

Durante essas pesquisas, os seguintes arquivos foram acessados:

APOCALIPSE4.txt -> 2 vezes ESTER9.txt -> 1 vez

ESTER1.txt -> 3 vezes APOCALIPSE2.txt -> 1 vez

APOCALIPSE7.txt -> 1 vez APOCALIPSE1.txt -> 1 vez

A figura 5 mostra os resultados dos Top 10 palavras mais buscadas e Top 10 arquivos mais acessados, exibidos em ordem crescente (esquerda) e decrescente (direita).

**Figura 5 – Resultados dos testes, top palavras e top páginas.**

Através da análise dos resultados obtidos durante os testes, foi possível concluir que os resultados apresentados através de uma pesquisa são precisos e as duas formas de ordenação funcionam. Além disso, os resultados dos Top-K também funcionaram adequadamente, portanto é possível assumir que o algoritmo é funcional e cumpre a proposta.

# 5. Conclusão

# Todos os requisitos propostos foram implementados com sucesso no programa em questão e funcionam adequadamente. Uma limitação observada no programa é que ele não funcionará corretamente caso o usuário digite duas palavras (não irá encontrar nada, mesmo que alguma dessas palavras se encontre em algum arquivo), fazer um tratamento para esses casos seria uma ótima possível implementação futura para o código. Além disso, o programa pode funcionar de forma inadequada com arquivos texto com codificações diferentes da UTF-8, já que a Java Virtual Machine (JVM) não consegue identificar acentos nesses tipos de arquivos.

**6. Bibliografia Consultada**

Goodrich, M. T. / Tamassia, R. (2013) “Estruturas de Dados e Algoritmos Em Java”, 5ª edição.

Horstman, C. S. / Cornell, G. (2009) “Core Java. Fundamentos – Volume 1”, 8ª edição.

Ricarte, I, L, M. (2001) “Programação Orientada a Objetos: Uma Abordagem com Java”, 1ª edição.

Goyvaerts, J. / Levithan, S. (2009) “Regular Expressions Cookbook”, 1ª edição.