UNIVERSIDADE FEDERAL DE JUIZ DE FORA
Departamento de Ciência da Computação - Curso Sistemas de Informação

Relatório:
Comércio Eletrônico

Trabalho apresentado ao curso de Sistemas de Informação, da Universidade Federal de Juiz de Fora, como parte dos requisitos necessários à aprovação na disciplina Estrutura de Dados II.

Docente: Vânia de Oliveira Neves

Discentes: Aline de Paula Sotte
Raiza Silva Campos
Thassya de Souza Abreu

Juiz de Fora
2017

Sumário

1 - Descrição das atividades realizadas por cada membro do grupo

- Aline:

Definição e desenvolvimento da interface; Criação e organização dos arquivos .txt

- Raiza:

Criação e atualização do relatório

- Thassya:

Pesquisa e definição das estruturas a serem utilizadas

- O desenvolvimento do trabalho foi feito com a colaboração de todas, em encontros do grupo, para melhor produtividade nas implementações das estruturas e testes. Definimos uma responsável por cada parte do trabalho a título de melhor organização, porém todas participaram de todas as atividades.

2 - Justificativa acerca da implementação escolhida

A árvore *Trie*, conhecida por ser boa para aplicações de dicionários e pesquisas textuais, foi nossa primeira opção para o trabalho. As informações que encontramos e a facilidade de implementação sustentaram nossa decisão.

Nos primeiros estudos, identificamos que com o método de busca por aproximação, conseguiríamos localizar dados semelhantes ao de uma palavra informada, a partir de sugestões, como seria necessário em casos de erro de digitação do usuário. Esta busca faz com que a *Trie* seja utilizada em corretores ortográficos e, em outras aplicações, como exemplo *browsers* e o Gmail.

Cogitamos, como base comparativa, a utilização da árvore patricia. Neste caso, as informações seriam armazenadas em seus contadores e ponteiros para sub-árvores e não em seus nódulos, como nas *Tries*. Esta árvore também é utilizada em corretores ortográficos e busca em documentos XML.

Suas principais vantagens são sua flexibilidade e escalabilidade para consultas e ser mais compacta do que as Tries, com isso, ganhando em velocidade. Em contrapartida, não possuem escalabilidade adequada quando se têm muitas chaves distintas para um mesmo caractere.

Uma vez tendo definido a *Trie*, analisamos qual abordagem seria a melhor dentre a *Trie Multiway* e a *Trie* Ternária. Sabendo que a primeira ocupa mais memória no armazenamento das palavras a serem buscadas por conter o alfabeto completo em cada nó da sua estrutura, escolhemos para implementação da *Trie* Ternária.

3 - Explicação sobre as estruturas de dados implementadas

As estruturas implementadas foram árvores *Tries* ternárias que pouco se diferem do tradicional algoritmo que vimos em sala. Vale destacar apenas que para ordenação dos nomes dos produtos, a árvore guarda um produto. Enquanto para ordenar por categorias, guarda um *arraylist* de produtos.

Como o nó foi implementado:

```
public class NoTrie {

    private char letra;
    private boolean ehFolha = false;
    private NoTrie esquerda;
    private NoTrie direita;
    private NoTrie meio;
    private Produto produto;
    private List<Produto> listaProduto = new ArrayList<>();
```

Na inserção, além de acrescentarmos o caractere correspondente no 'No', como seria o normal de uma *Trie*, quando está no final, é inserido um produto - no caso da inserção da *Trie* de produtos – e, para a *Trie* de categorias, o produto é adicionado no *ArrayList*.

Quando encontramos o último nó a ser guardado, já inserimos o produto e marcamos como folha. Foi necessário apenas adicionar estes dois métodos:

```
public void setProduto(Produto produto) {
    this.produto = produto;
    this.ehFolha = true;
}

public void addItemListaProduto(Produto p){
    this.listaProduto.add(p);
    this.ehFolha=true;
}
```

A função de *autocomplete* utiliza as chaves armazenadas na *Trie* Ternária para fazer a busca, esta se difere no produto e na categoria.

Na categoria, quando o usuário busca uma palavra que existe na *trie* é retornado para ele o *arraylist* de produtos salvos para aquela busca. Se o usuário digita alguma palavra que tem semelhança inserida na trie, mesmo que com erros, é feita uma busca até o último caractere encontrado na Trie. A partir deste nó é feita uma busca até as folhas retornando todos os produtos encontrados neste caminho. Como categorias costuma ter um número de elementos não muito grande, mesmo que o usuário erre algo na palavra buscada retornará a categoria que ele deseja ou as categorias que têm o

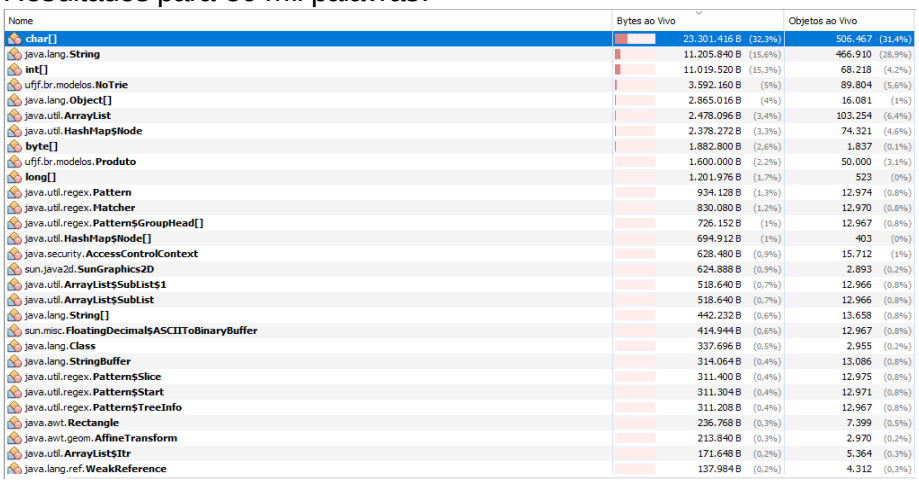
mesmo prefixo em comum. Se a busca do usuário não encontrar nenhuma semelhança com a Trie construída é retornada todas as categorias cadastradas na Trie.

Para o produto, quando o usuário digita um termo que existe na *Trie,* ele é retornado e é realizada uma busca por semelhantes até o final da árvore. Se o produto não existir, ele tenta encontrar semelhanças da mesma forma que a categoria.

Já para a estrutura de ordenação, utilizamos o *MergeSort*, pois, apesar de consumir mais memória em comparação aos demais, ele é de fácil implementação, é o mais indicado para ordenações em árvores, além de ser confiável e estável. A única adaptação feita em relação ao algoritmo visto em sala de aula foi utilizar um *ArrayList* de Produtos para fazer a comparação com os itens requeridos, como Nome do produto, Categoria e Preço, ao invés de um vetor de inteiros.

4 - Análise de desempenho

Resultados para 50 mil palavras:

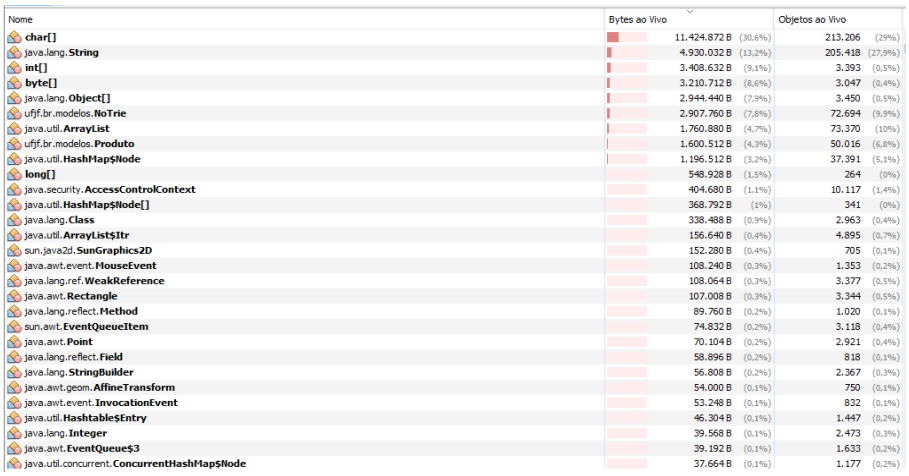


Nome	Bytes ao Vivo	Objetos ao Vivo
char[]	23.301.416 B (32,3%)	506.467 (31,4%)
java.lang.String	11.205.840 B (15,6%)	466.910 (28,9%)
int[]	11.019.520 B (15,3%)	68.218 (4,2%)
ufjf.br.modelos.NoTrie	3.592.160 B (5%)	89.804 (5,6%)
java.lang.Object[]	2.865.016 B (4%)	16.081 (1%)
java.util.ArrayList	2.478.096 B (3,4%)	103.254 (6,4%)
java.util.HashMap$Node	2.378.272 B (3,3%)	74.321 (4,6%)
byte[]	1.882.800 B (2,6%)	1.837 (0,1%)
ufjf.br.modelos.Produto	1.600.000 B (2,2%)	50.000 (3,1%)
long[]	1.201.976 B (1,7%)	523 (0%)
java.util.regex.Pattern	934.128 B (1,3%)	12.974 (0,8%)
java.util.regex.Matcher	830.080 B (1,2%)	12.970 (0,8%)
java.util.regex.Pattern$GroupHead[]	726.152 B (1%)	12.967 (0,8%)
java.util.HashMap$Node[]	694.912 B (1%)	403 (0%)
java.security.AccessControlContext	628.480 B (0,9%)	15.712 (1%)
sun.java2d.SunGraphics2D	624.888 B (0,9%)	2.893 (0,2%)
java.util.ArrayList$SubList$1	518.640 B (0,7%)	12.966 (0,8%)
java.util.ArrayList$SubList	518.640 B (0,7%)	12.966 (0,8%)
java.lang.String[]	442.232 B (0,6%)	13.658 (0,8%)
sun.misc.FloatingDecimal$ASCIIToBinaryBuffer	414.944 B (0,6%)	12.967 (0,8%)
java.lang.Class	337.696 B (0,5%)	2.955 (0,2%)
java.lang.StringBuffer	314.064 B (0,4%)	13.086 (0,8%)
java.util.regex.Pattern$Slice	311.400 B (0,4%)	12.975 (0,8%)
java.util.regex.Pattern$Start	311.304 B (0,4%)	12.971 (0,8%)
java.util.regex.Pattern$TreeInfo	311.208 B (0,4%)	12.967 (0,8%)
java.awt.Rectangle	236.768 B (0,3%)	7.399 (0,5%)
java.awt.geom.AffineTransform	213.840 B (0,3%)	2.970 (0,2%)
java.util.ArrayList$Itr	171.648 B (0,2%)	5.364 (0,3%)
java.lang.ref.WeakReference	137.984 B (0,2%)	4.312 (0,3%)

Resultados para 250 mil palavras:

Nome	Bytes ao Vivo	Objetos ao Vivo
char[]	52.464.736 B (30,3%)	1.425.918 (31,2%)
java.lang.String	31.941.912 B (18,5%)	1.330.913 (29,2%)
int[]	22.083.592 B (12,8%)	160.894 (3,5%)
ufjf.br.modelos.NoTrie	18.037.320 B (10,4%)	450.933 (9,9%)
java.util.ArrayList	11.588.688 B (6,7%)	482.862 (10,6%)
java.lang.Object[]	10.068.928 B (5,8%)	34.270 (0,8%)
ufjf.br.modelos.Produto	8.000.000 B (4,6%)	250.000 (5,5%)
java.util.regex.Pattern	2.270.808 B (1,3%)	31.539 (0,7%)
java.util.regex.Matcher	2.018.304 B (1,2%)	31.536 (0,7%)
java.util.regex.Pattern$GroupHead[]	1.765.848 B (1%)	31.533 (0,7%)
byte[]	1.489.032 B (0,9%)	1.181 (0%)
java.util.ArrayList$SubList$1	1.261.280 B (0,7%)	31.532 (0,7%)
java.util.ArrayList$SubList	1.261.280 B (0,7%)	31.532 (0,7%)
java.lang.String[]	1.035.744 B (0,6%)	32.184 (0,7%)
sun.misc.FloatingDecimal$ASCIIToBinaryBuffer	1.009.056 B (0,6%)	31.533 (0,7%)
java.lang.StringBuffer	760.608 B (0,4%)	31.692 (0,7%)
java.util.regex.Pattern$Slice	756.960 B (0,4%)	31.540 (0,7%)
java.util.regex.Pattern$Start	756.864 B (0,4%)	31.536 (0,7%)
java.util.regex.Pattern$TreeInfo	756.792 B (0,4%)	31.533 (0,7%)
java.util.HashMap$Node	689.760 B (0,4%)	21.555 (0,5%)
java.lang.Class	337.432 B (0,2%)	2.953 (0,1%)

Resultados para 500 mil palavras (lowercase):



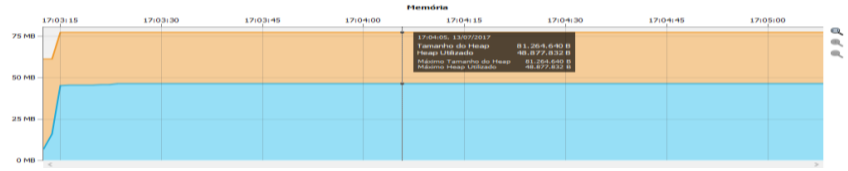
Nome	Bytes ao Vivo	Objetos ao Vivo
char[]	11.424.872 B (30,6%)	213.206 (29%)
java.lang.String	4.930.032 B (13,2%)	205.418 (27,9%)
int[]	3.408.632 B (9,1%)	3.393 (0,5%)
byte[]	3.210.712 B (8,6%)	3.047 (0,4%)
java.lang.Object[]	2.944.440 B (7,9%)	3.450 (0,5%)
ufjf.br.modelos.NoTrie	2.907.760 B (7,8%)	72.694 (9,9%)
java.util.ArrayList	1.760.880 B (4,7%)	73.370 (10%)
ufjf.br.modelos.Produto	1.600.512 B (4,3%)	50.016 (6,8%)
java.util.HashMap$Node	1.196.512 B (3,2%)	37.391 (5,1%)
long[]	548.928 B (1,5%)	264 (0%)
java.security.AccessControlContext	404.680 B (1,1%)	10.117 (1,4%)
java.util.HashMap$Node[]	368.792 B (1%)	341 (0%)
java.lang.Class	338.488 B (0,9%)	2.963 (0,4%)
java.util.ArrayList$Itr	156.640 B (0,4%)	4.895 (0,7%)
sun.java2d.SunGraphics2D	152.280 B (0,4%)	705 (0,1%)
java.awt.event.MouseEvent	108.240 B (0,3%)	1.353 (0,2%)
java.lang.ref.WeakReference	108.064 B (0,3%)	3.377 (0,5%)
java.awt.Rectangle	107.008 B (0,3%)	3.344 (0,5%)
java.lang.reflect.Method	89.760 B (0,2%)	1.020 (0,1%)
sun.awt.EventQueueItem	74.832 B (0,2%)	3.118 (0,4%)
java.awt.Point	70.104 B (0,2%)	2.921 (0,4%)
java.lang.reflect.Field	58.896 B (0,2%)	818 (0,1%)
java.lang.StringBuilder	56.808 B (0,2%)	2.367 (0,3%)
java.awt.geom.AffineTransform	54.000 B (0,1%)	750 (0,1%)
java.awt.event.InvocationEvent	53.248 B (0,1%)	832 (0,1%)
java.util.Hashtable$Entry	46.304 B (0,1%)	1.447 (0,2%)
java.lang.Integer	39.568 B (0,1%)	2.473 (0,3%)
java.awt.EventQueue$3	39.192 B (0,1%)	1.633 (0,2%)
java.util.concurrent.ConcurrentHashMap$Node	37.664 B (0,1%)	1.177 (0,2%)

Resultados para 500 mil palavras:

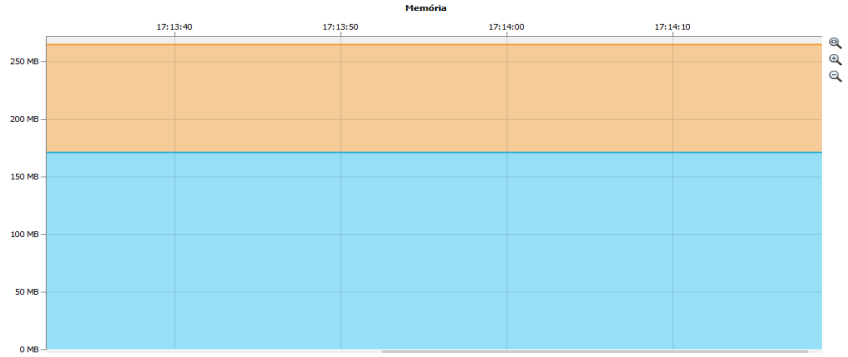
Nome	Bytes ao Vivo	Objetos ao Vivo
ufjf.br.modelos.NoTrie	67.254.440 B (28,2%)	1.681.361 (25,4%)
char[]	54.246.184 B (22,8%)	1.351.985 (20,5%)
java.util.ArrayList	40.469.160 B (17%)	1.686.215 (25,5%)
java.lang.String	31.815.096 B (13,4%)	1.325.629 (20,1%)
int[]	13.157.504 B (5,5%)	23.957 (0,4%)
ufjf.br.modelos.Produto	12.713.568 B (5,3%)	397.299 (6%)
java.lang.Object[]	11.365.088 B (4,8%)	7.186 (0,1%)
byte[]	1.462.040 B (0,6%)	1.082 (0%)
java.util.HashMap$Node	602.560 B (0,3%)	18.830 (0,3%)
java.lang.Class	337.328 B (0,1%)	2.952 (0%)
java.util.regex.Pattern	323.856 B (0,1%)	4.498 (0,1%)
java.util.regex.Matcher	287.680 B (0,1%)	4.495 (0,1%)
java.lang.StringBuilder	283.704 B (0,1%)	11.821 (0,2%)
long[]	258.528 B (0,1%)	163 (0%)
java.util.regex.Pattern$GroupHead[]	251.552 B (0,1%)	4.492 (0,1%)
java.security.AccessControlContext	226.640 B (0,1%)	5.666 (0,1%)
java.util.HashMap$Node[]	200.912 B (0,1%)	318 (0%)
java.util.ArrayList$SubList$1	179.680 B (0,1%)	4.492 (0,1%)
java.util.ArrayList$SubList	179.680 B (0,1%)	4.492 (0,1%)
java.lang.String[]	170.264 B (0,1%)	5.141 (0,1%)
java.util.ArrayList$Itr	165.184 B (0,1%)	5.162 (0,1%)

5 - Análise de gasto de memória

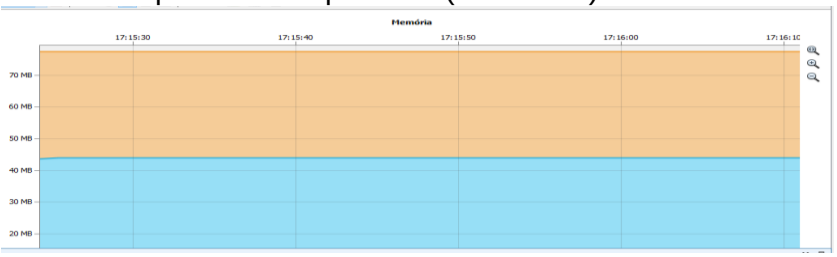
Resultados para 50 mil palavras:



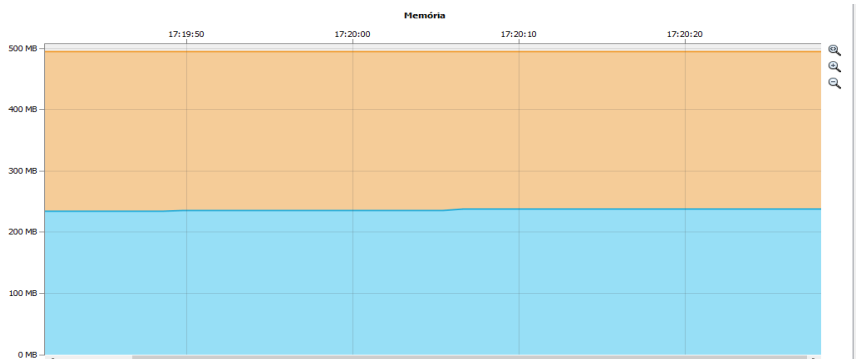
Resultados para 250 mil palavras:



Resultados para 500 mil palavras (lowercase):



Resultados para 500 mil palavras:



6 - Referências

BUENO, M. 2016. **Árvores Trie e Patricia**. Disponível em: <https://marciobueno.com/arquivos/ensino/ed2/ED2_06_Trie_Patricia.pdf>. Acesso em: 10 jul. 2017.

GIACON, A. P. *et al*. **Merge Sort**. Disponível em: <http://www.ft.unicamp.br/liag/siteEd/includes/arquivos/MergeSortResumo_Grupo4_ST364A_2010.pdf>. Acesso em 13 jul. 2017.

MACHADO, C. 2008. **Árvores Patricia**. Disponível em: <http://www.inf.ufrgs.br/~cagmachado/INF01124/t3.htm>. Acesso em: 10 jul. 2017.