

UNIVERSIDADE FEDERAL DE JUIZ DE FORA  
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO  
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MATEUS COUTINHO MARIM

**CONTADOR DE FREQUÊNCIA DE PALAVRAS**

TRABALHO DE ESTRUTURAS DE DADOS II

JUIZ DE FORA  
2016

MATEUS COUTINHO MARIM

## **CONTADOR DE FREQUÊNCIA DE PALAVRAS**

Trabalho da matéria Estruturas de  
Dados II do curso de Ciência da  
Computação da Universidade Federal  
de Juiz de Fora.

Professor: Jairo Francisco de Souza

JUIZ DE FORA

2016

## Sumário

1. INTRODUÇÃO.....	4
2. TAD IMPLEMENTADO.....	6
2.1 REPRESENTAÇÃO DO TAD.....	7
2.2 ANÁLISE DA COMPLEXIDADE NA INSERÇÃO.....	8
2.3 ORDENAÇÃO LEXICOGRÁFICA.....	9
2.3.1 ANÁLISE DA COMPLEXIDADE DO TREESORT.....	10
2.4 ORDENAÇÃO POR FREQUÊNCIA DAS CHAVES.....	10
2.4.1 ANÁLISE DA COMPLEXIDADE DO MERGESORT.....	11
2.5 ANÁLISE DA COMPLEXIDADE DA SOLUÇÃO.....	12
2.5.1 ANÁLISE EXPERIMENTAL DO DESEMPENHO.....	13
2.5.2 ANÁLISE DO GASTO DE MEMÓRIA.....	17
2.5.3 ANÁLISE EXPERIMENTAL DO GASTO DE MEMÓRIA.....	18
3. MÉTODO PARA IMPRIMIR PALAVRAS MAIS FREQUENTES DISCRIMINADAS POR ARQUIVO.....	20
4. COMO USAR O PROGRAMA.....	21
5. CONCLUSÃO.....	23

# 1. INTRODUÇÃO

Contar a frequência em textos muitas vezes se apresenta como um problema banal que pode ser facilmente resolvido, mas se não forem utilizados bons métodos para a resolução deste problema os custos para se chegar no resultado podem ser inviáveis, sendo estes custos de desempenho e gastos de memória.

O objetivo deste trabalho é encontrar uma estratégia eficiente para se contar a frequência de palavras em textos planos sem formatação combinando um bom desempenho com o menor gasto de memória possível.

Descrição do problema:

Dado um arquivo texto, vamos montar a lista de palavras que ocorrem no texto, com suas respectivas frequências, em ordem decrescente de frequência. Por exemplo, suponha o texto abaixo:

*We must not underrate the gravity of the task which lies before us or the temerity of the ordeal, to which we shall not be found unequal. We must expect many disappointments, and many unpleasant surprises, but we may be sure that the task which we have freely accepted is one not beyond the compass and the strength of the British Empire and the French Republic.*

A saída deve ser:

the 9  
and 3  
not 3  
of 3  
we 3  
which 3  
We 2  
be 2  
many 2  
must 2

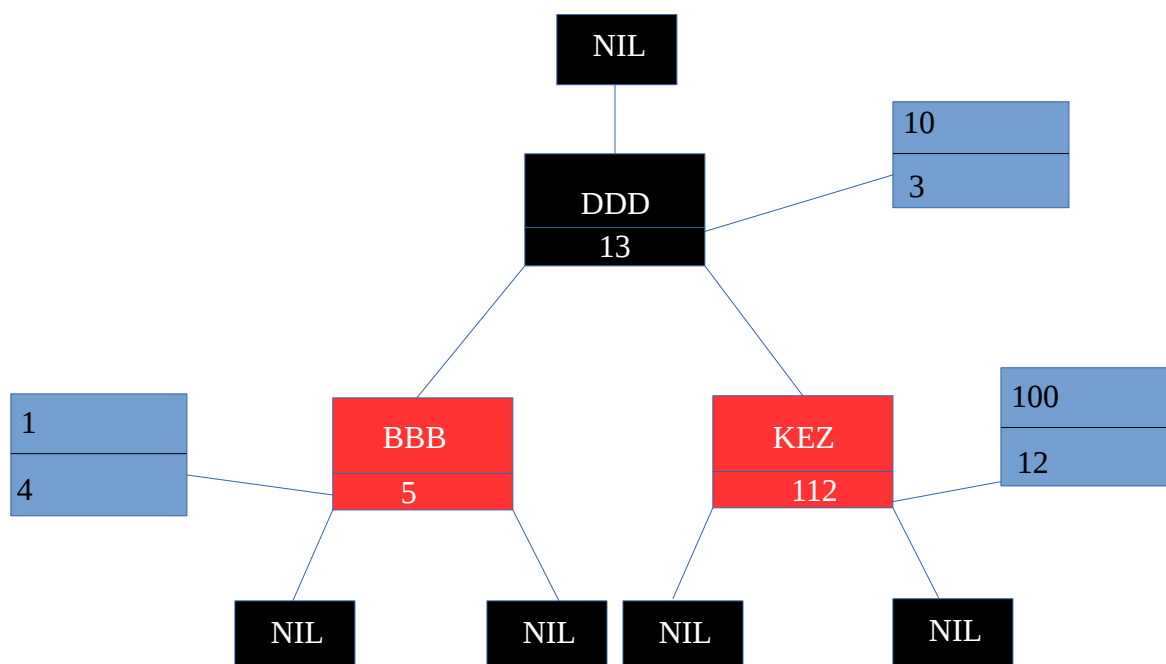
## **2. TAD IMPLEMENTADO**

O TAD escolhido para armazenar as palavras e contar as frequência foi uma árvore vermelho e preta, pois era necessário escolher uma estrutura de dados com uma maior flexibilidade na inserção para que se caísse menos vezes no pior caso da estrutura de  $\Omega(2 \cdot \log_2(N))$  e no final tivéssemos o desempenho das árvores binárias de busca, na ordem de  $O(\log_2(N))$ .

Como a árvore vermelho e preta na sua implementação mais simples apenas permite a contagem de frequência se fossem permitidas chaves duplicadas, foi necessário pensar em um jeito melhor para contar as frequências sem que precisasse percorrer toda a árvore para se obter o valor final das frequências.

Para resolver este problema a estrutura foi aumentada adicionando um contador nos nós da árvore para contar a frequência global e um vetor de frequências locais para cada arquivo processado, para que, toda vez que na inserção de árvore binária se chegasse em um nó com a mesma chave do que está sendo inserido, ao invés de se adicionar um nó repetido, a frequência global e a local que está relacionada ao arquivo do qual a chave veio iriam ser incrementadas e a inserção seria terminada, evitando a adição de duplicatas.

## 2.1 REPRESENTAÇÃO DO TAD



*Figura 1. Representação da estrutura utilizada com entrada de 2 arquivos.*

A estrutura na figura é uma árvore vermelho e preta em que seus nós além de armazenar uma palavra, também contém um campo com a frequência global da palavra associada e um vetor que armazena a frequência local da palavra em relação a cada arquivo de entrada.

## 2.2 ANÁLISE DA COMPLEXIDADE NA INSERÇÃO

Como não existem nós duplicados na árvore podemos dizer que seu tamanho vai ser a quantidade de palavras únicas existentes nos textos processados ao qual vamos chamar de  $P$ , como vão ser feitas  $N$  inserções e a cada inserção vai haver incremento das frequências locais e globais com complexidade  $O(1)$  dando no total uma complexidade de  $O(N)$ , o pior caso da inserção das chaves na estrutura vai passar a ser

$$T(N) = 2 * N * \log_2(P) + N = N * (2 * \log_2(P) + 1) \text{ .}$$

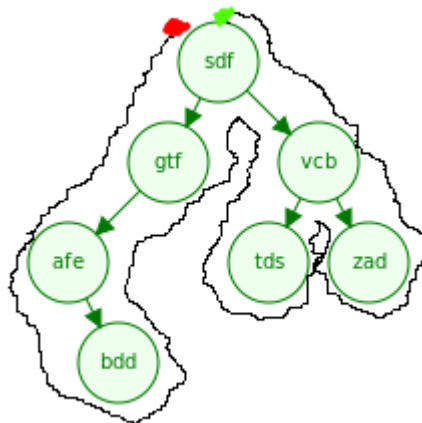
$$\Omega(N * \log_2(P))$$



## 2.3 ORDENAÇÃO LEXICOGRÁFICA

O próximo passo para resolver o problema e apresentá-lo como pedido, foi fazer uma ordenação que deixasse as chaves em ordem alfabética, para isso nos aproveitamos das propriedades das árvores binárias de busca e executamos uma transversal In-order na árvore que de acordo com as propriedades vai imprimir os nós da árvore ordenados, como estamos lidando com strings, elas vão sair em ordem alfabética e vão ser armazenadas em uma lista.

Este procedimentos citados acima são utilizados pelo algoritmo treeSort representado na figura 2, onde a bolinha vermelha marca o início da caminhada e a verde marca o fim.



*Figura 2. Representação do treeSort com início na bola vermelha e fim na verde, o resultado nessa árvore vai ser {bdd, afe, gtf, sdf, vcb, tds, zad}.*

### 2.3.1 ANÁLISE DA COMPLEXIDADE DO TREESORT

O algoritmo treeSort tem como complexidade no pior caso para ordenar uma árvore binária balanceada  $\Omega(N * m * \log_2(N))$ ,  $m$  é o tamanho médio de uma string, supondo que fosse necessário construir a árvore com  $N$  nós, como já temos a árvore binária construída, balanceada e com tamanho  $P$ , então o algoritmo vai ter como complexidade no pior caso apenas  $\Omega(m * P)$ .

## 2.4 ORDENAÇÃO POR FREQUÊNCIA DAS CHAVES

Chegamos na parte em que temos que fazer com que a saída mostre as chaves ordenadas pela maior frequência, as que tiverem as mesmas frequências vão ser mostradas em ordem alfabética.

Para isso precisamos utilizar um algoritmo que depois de ordenar os nós pela frequência das chaves as mantenha na ordem que apareceram antes de serem ordenadas, esse tipo de algoritmo de ordenação é chamado de ordenação estável.

O algoritmo que iremos escolher é o mergeSort que satisfaz os requisitos de ser estável e ter uma complexidade aceitável para a nossa necessidade.

Exemplo da execução do mergeSort:

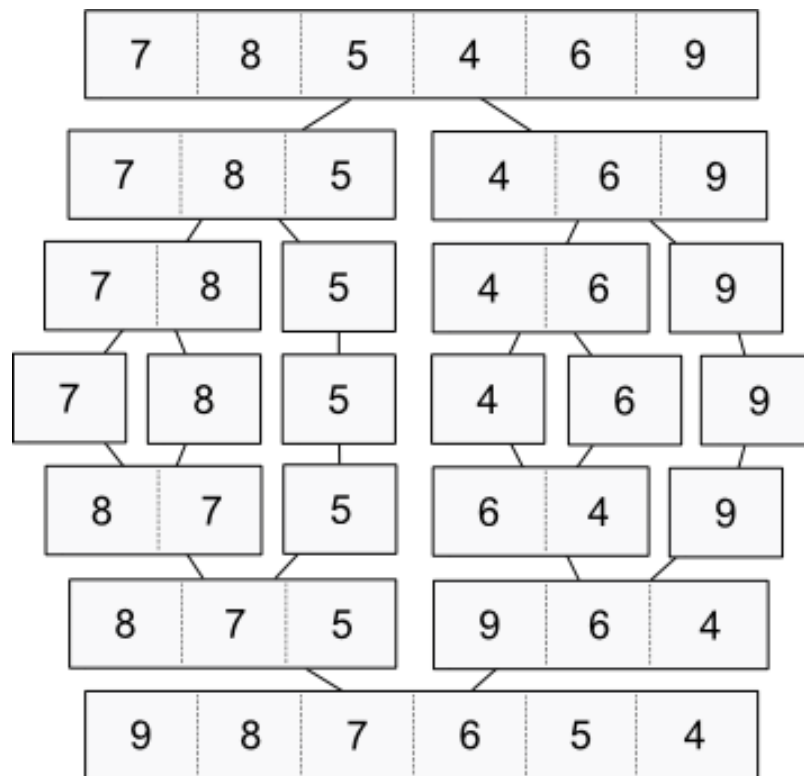


Figura 3: Árvore de execução do algoritmo mergeSort.

### 2.4.1 ANÁLISE DA COMPLEXIDADE DO MERGESORT

O algoritmo mergeSort tem como complexidade de ordenação no pior caso  $\Omega(N \cdot \log_2(N))$  pois ele vai quebrando o vetor inicial em dois a cada chamada recursiva, gerando uma árvore binária de recursividade de altura  $\log_2(N)$  e essa divisão é feita  $N$  vezes como mostrado na figura 3, como a nossa lista a ser ordenada tem tamanho  $P$  a complexidade no pior caso vai ser  $\Omega(P \cdot \log_2(P))$ .

## 2.5 ANÁLISE DA COMPLEXIDADE DA SOLUÇÃO

Seguindo os passos anteriores o problema do Contador de Frequência de Palavras é resolvido, mas ainda precisamos fazer a análise do desempenho da solução para verificar se os métodos utilizados são viáveis e satisfazem o requisito de ter um bom desempenho na execução.

Para fazermos esta análise primeiro vai ser feita a análise assintótica da solução e logo depois vão ser feitos experimentos suficientes para verificar se a curva gerada pelos experimentos batem com a ordem de complexidade achada para os procedimentos, caso sejam curvas equivalentes, pode-se dizer que a solução tem o desempenho esperado.

Primeiro é feita uma função  $T(N,P)$  que descreve o comportamento assintótico da solução com  $N$  palavras e  $P$  palavras únicas:

$$T(N,P) = 2 * N * m * \log_2(P) + P * m + P * m * \log_2(P)$$

A fórmula acima é definida como a soma das fórmulas que representam a complexidade dos métodos utilizados na solução e com ela podemos isolar o termo de maior grau para termos a complexidade da solução:

$$T(N,P) = 2 * N * m * \log_2(P)$$

$$\Omega(N * m * \log_2(P))$$

Portanto, temos que a complexidade da solução no pior caso é  $\Omega(N * m * \log_2(P))$ .

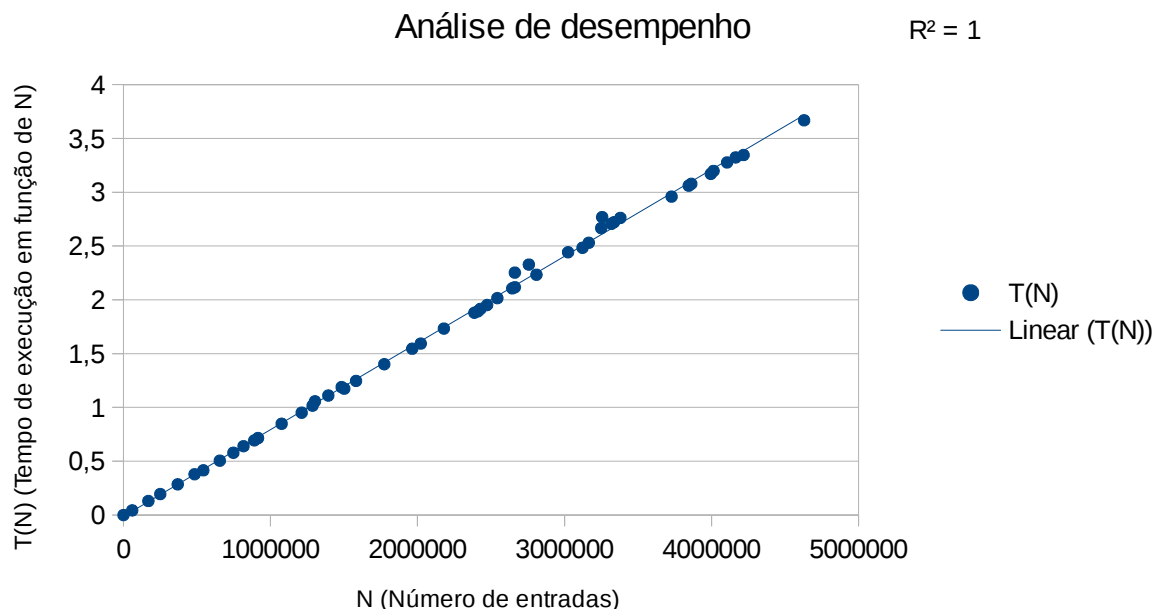
### **2.5.1 ANÁLISE EXPERIMENTAL DO DESEMPENHO**

Foram realizados 50 experimentos para gerar a curva representando o tempo de execução em função de N entradas, segue abaixo uma tabela com os resultados:

N	Palavras Unicas	T(N)
0	0	0,000001
59462	5379	0,043211
169354	14670	0,130386
249907	18746	0,194553
368928	21699	0,285219
483508	24149	0,378919
543105	29287	0,415203
655133	32335	0,505177
747668	35718	0,57853
816635	37847	0,639832
890102	39298	0,693739
915064	40301	0,715122
1075737	43086	0,848166
1212071	45964	0,951114
1286068	47361	1,01622
1302610	47802	1,05656
1393554	48862	1,11079
1483745	49638	1,18852
1501995	50002	1,17432
1581952	52251	1,2466
1774119	56975	1,40155
1964047	61177	1,54578
2022217	61661	1,59337
2179561	63263	1,73294
2386924	67950	1,87979
2411242	68357	1,89405
2428483	68849	1,91587
2473429	69326	1,9516
2543160	70093	2,01562
2644725	71671	2,10668
2662161	71718	2,11694
2662227	71718	2,25291
2757538	72487	2,32786
2809275	73882	2,23339
3024311	76166	2,44223
3122782	77285	2,48417
3165293	77750	2,53048
3249565	79097	2,66634
3256138	79177	2,76852
3318608	79522	2,70648
3335336	79574	2,72008
3380451	80635	2,76132
3728459	89887	2,95935
3845363	91736	3,06086
3862800	92059	3,07829
3995288	92783	3,17083
4014045	92966	3,19805
4105783	94120	3,27673
4164664	94579	3,32331
4218248	95285	3,34623
4629933	97510	3,66846

*Tabela 1: Resultados dos experimentos.*

E abaixo está o gráfico que representa os resultados:

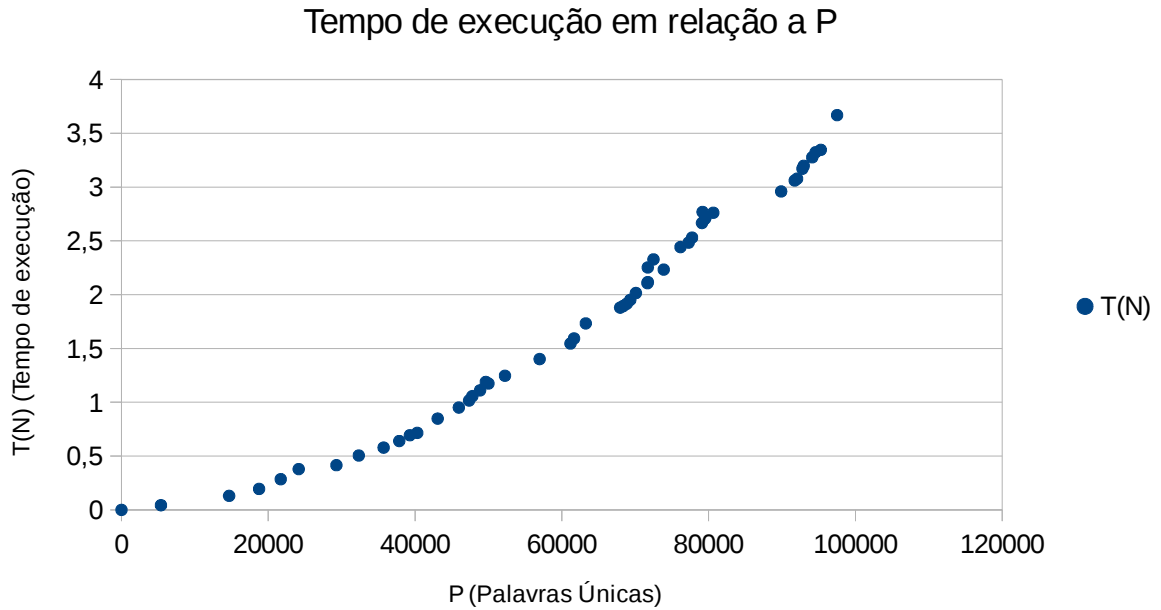


Como pode ser visto o gráfico gerou uma curva linear, e para confirmar basta ver que o valor do coeficiente de determinação  $R^2$  é muito próximo a 1 (o valor foi arredondado, mas  $R^2=0,998505$ ), isso quer dizer que a equação linear gerada pelo gráfico consegue prever quase 100% dos resultados.

Esse comportamento linear consegue ser explicado pelo fato de que o valor da variável P cresce muito pouco conforme o valor de N cresce, isso se explica pelo “Princípio de Pareto”, ou mais comumente, a lei “os 80/20”, que é uma relação que descreve casualidades e resultados. Este princípio alega que aproximadamente 80% da saída é um resultado direto de 20% da entrada.

O “Princípio de Pareto” é utilizado para explicar o comportamento da curva pois ele é facilmente observado na frequência do uso de palavras nas linguagens de todo o mundo, incluindo a língua inglesa que é a língua das palavras da

entrada do programa, portanto ele se aplica ao problema.



*Gráfico 2: Tempo de execução em relação a quantidade de palavras únicas.*

Portanto, quanto maior a entrada, maior a diferença entre  $N$  e  $P$ , fazendo com que na maior parte dos casos da inserção apenas uma pesquisa seja feita e o contador do nó seja incrementado, fazendo com que a curva tenda a ser linear, se considerarmos que  $P$  com uma entrada muito grande se torne constante, então a complexidade no pior caso para grandes quantidades de dados vai ser de  $\Omega(N*m)$ .



## 2.5.2 ANÁLISE DO GASTO DE MEMÓRIA

Para analisar o gasto de memória é necessário somar a complexidade da memória usada em cada passo da solução, a inserção na árvore vermelho e preto tem complexidade de memória no pior caso de  $O(P)$  mais  $O(P*M)$ , onde  $M$  é o tamanho do vetor associado a cada nó da árvore, a ordenação treeSort  $O(P)$  o mergeSort tem  $O(P)$  mais  $O(P)$  auxiliar durante a execução, a complexidade do gasto de memória vai ser  $G(P)=P*(3+M)$  no pior caso, logo a complexidade de gasto de memória vai ser  $O(P*M)$ .

Como o gasto de memória vai depender principalmente do crescimento da quantidade de palavras únicas  $P$ , o crescimento da curva do gasto de memória vai ser proporcional ao crescimento de  $P$  em relação a  $N$ , como mostrado no Gráfico 3 e comprovado na análise experimental.

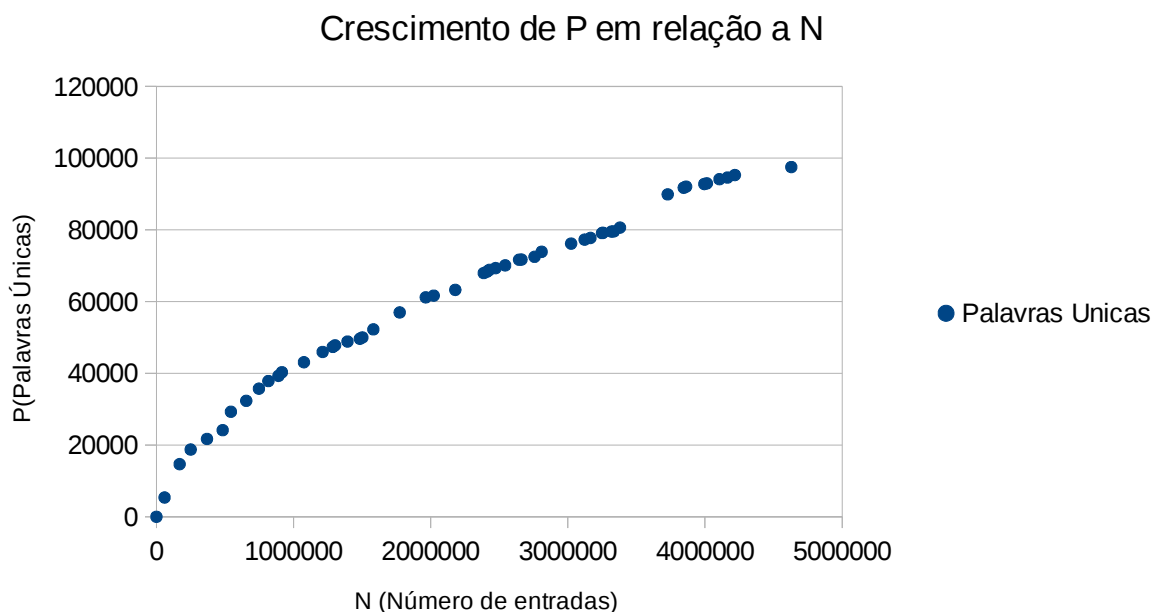


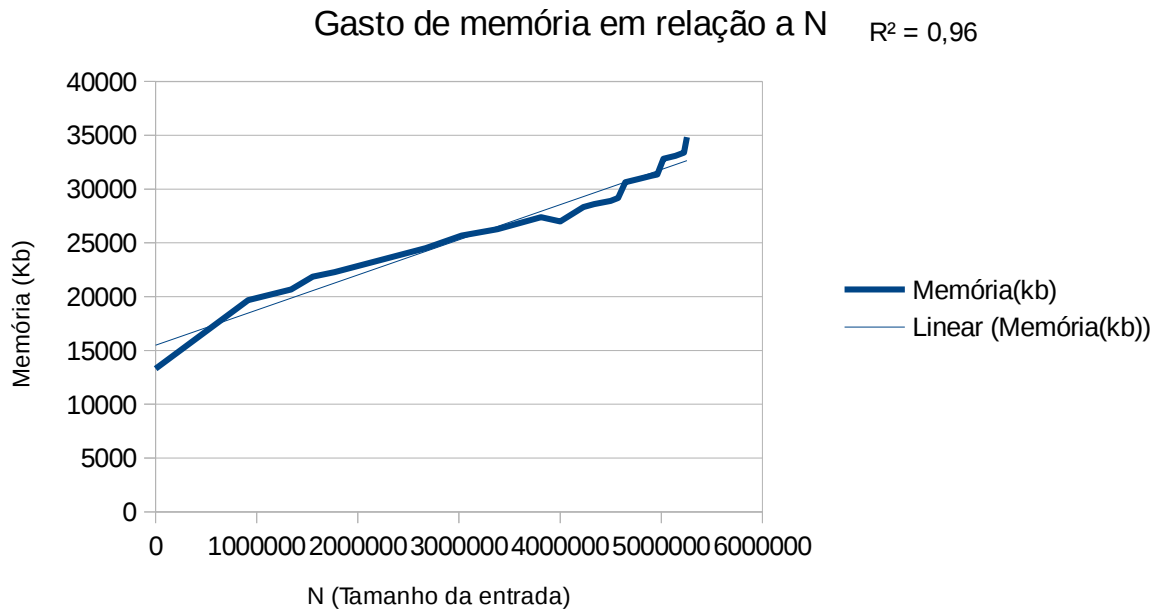
Gráfico 3: Crescimento de  $P$  em relação a  $N$ .

### 2.5.3 ANÁLISE EXPERIMENTAL DO GASTO DE MEMÓRIA

Foram realizados 22 experimentos para gerar a curva representando o gasto de memória do programa, segue abaixo uma tabela com os resultados:

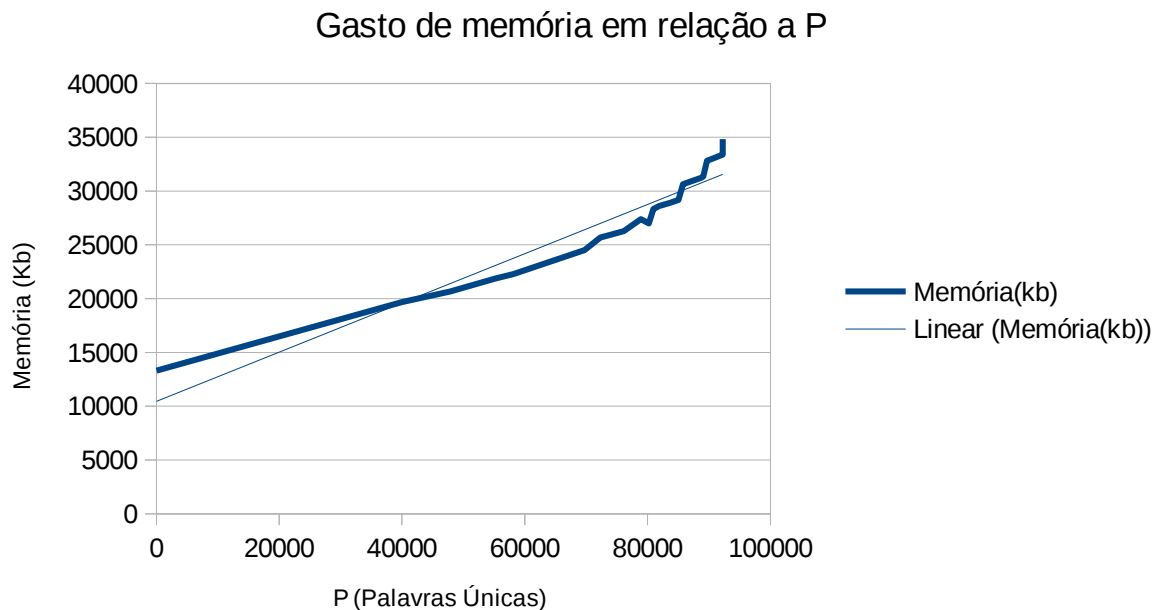
P	N	Memória(kb)
0	0	13316
39995	915303	19672
47905	1339288	20680
55152	1554159	21852
58135	1769370	22292
69748	2671885	24516
72312	3025226	25680
76109	3382950	26268
78892	3811768	27400
80227	4000768	26992
80934	4229359	28324
81842	4336755	28604
83654	4500164	28892
85020	4573941	29184
85758	4645074	30648
87233	4773318	30940
88722	4899517	31224
89086	4961585	31364
89685	5021055	32824
90955	5139655	33100
92228	5223005	33388
92228	5253868	34836

E abaixo está o gráfico que representa os resultados:



*Gráfico 4: Gasto de memória em relação a N*

Como é possível observar no gráfico, o gasto de memória tem uma taxa de crescimento maior com uma entrada menor, isso também se deve pelo Princípio de Pareto, pois após uma certa quantidade de inserções as palavras que aparecem mais frequentemente na língua inglesa se repetem mais, fazendo com que o gasto de memória aumente, mas sendo por causa do incremento dos contadores e por causa de inserções de palavras mais específicas do texto.

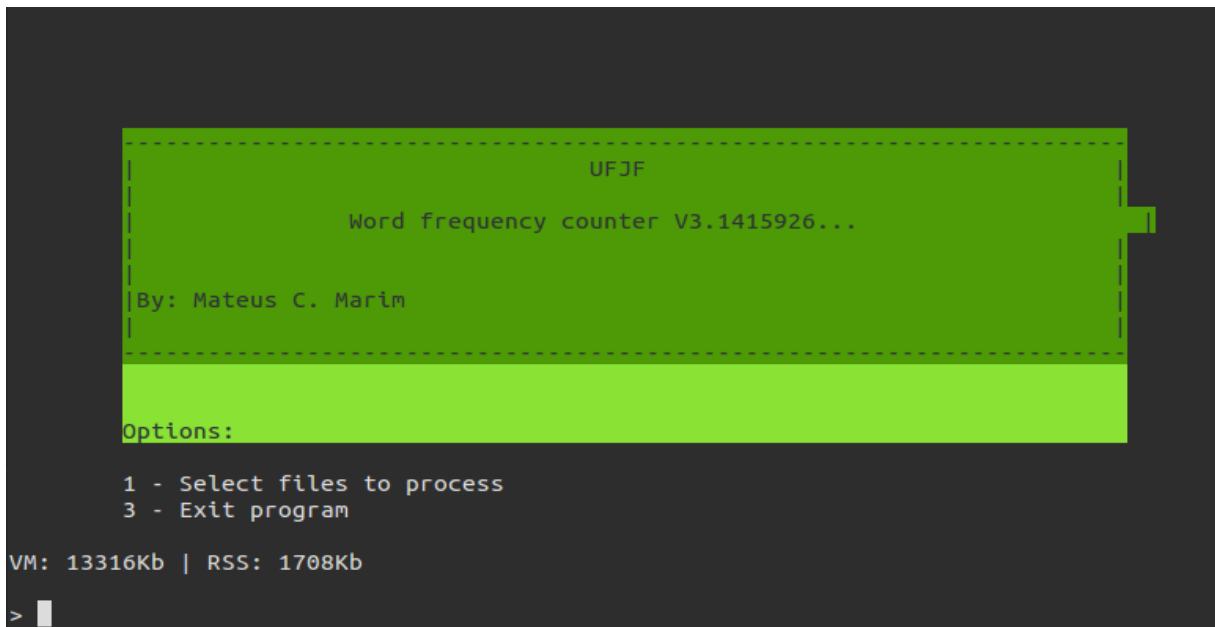


### 3. MÉTODO PARA IMPRIMIR PALAVRAS MAIS FREQUENTES DISCRIMINADAS POR ARQUIVO

Para imprimir as palavras mais frequentes discriminadas por arquivo foi necessário percorrer o vetor com os nós ordenados da árvore salvando os nós que tivessem frequência local no arquivo desejado maior que zero, mas como estavam ordenados em relação a frequência global, a ordenação lexicográfica e por frequência não foram mantidas, então foi necessário rodar o mergeSort uma vez para ordenar os nós lexicograficamente e outra para ordená-los por frequência, restaurando as propriedades desejadas com uma complexidade  $O(P)$  .

## 4. COMO USAR O PROGRAMA

Após executar o programa e colocar todos os arquivos para serem processados dentro da pasta Input, a primeira coisa que se deve fazer para liberar todas opções é selecionar a opção 1 para selecionar os arquivos para serem processados.



```
UFJF
Word frequency counter V3.1415926...
By: Mateus C. Marim

Options:
1 - Select files to process
3 - Exit program

VM: 13316Kb | RSS: 1708Kb
> 
```

Após aparecer na tela os arquivos que foram detectados na pasta de Input deve-se pressionar y caso se deseje continuar e n caso contrário.

```
b12.txt
a2.txt
a1.txt
b1.txt
pg18500.txt
b11.txt
a8.txt
b4.txt
b6.txt
a5.txt
a7.txt
a4.txt
b5.txt
a9.txt
b8.txt
b3.txt
b7.txt
b10.txt
b9.txt
b2.txt
a10.txt
```

23 files found...

Do you want to proceed? (y/n): █

Após os arquivos serem processados, todas as opções estarão liberadas, depois basta fazer o que for pedido em cada tela do programa.

```
File Edit View Search Terminal Help
UFJF
Word frequency counter V3.1415926...
By: Mateus C. Marim
Options:
1 - Select files to process
2 - X more frequent words in specific DB
3 - X more frequent words in all DB
4 - All words that occur only one time
5 - Display the name of the files processed
6 - Exit program
DB loaded in 4.17404 seconds.
VM: 34836Kb | RSS: 24864Kb
> █
```

## 5. CONCLUSÃO

A estrutura implementada satisfaz os requisitos de resolver o problema apresentado em um tempo bom e com um gasto de memória aceitável, a solução poderia ter sido melhor resolvida se tivesse sido usada uma árvore trie, mas como o programa utilizado já havia sido todo implementado não haveria tempo hábil para reescrever todo código utilizando essa estrutura.