

# Trabalho Prático 0

## Frequência de Palavras

Gilson Urbano Ferreira Dias - 2015430959  
guf@ufmg.br

### 1. Introdução

Frequência de palavras é um problema de Pesquisa Digital; a pesquisa digital é baseada na representação das chaves como uma sequência de caracteres ou de dígitos. Os métodos de pesquisa digital são vantajosos quando o tamanho do dicionário é grande ou de tamanho dinâmico, outro aspecto interessante e que será desenvolvido ao longo do trabalho é a possibilidade de utilização da pesquisa digital para localizar a ocorrência de um grupo de palavras em um texto, isso com tempo logarítmico em relação ao tamanho do texto.

A estrutura de dados proposta a ser utilizada foi uma Trie, a Trie foi definida em 1960 por Edward Fredkin, e o nome vem de Retrieval, relacionado à recuperação de informação. A trie é uma árvore M-ária cujos nós são vetores de M componentes com campos correspondentes aos dígitos ou caracteres que formam chaves. No caso específico desse trabalho, os M componentes serão letras que fazem parte do alfabeto latino padrão ISO que consiste em 26 letras a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, todas minúsculas. Cada nó no nível i representa o conjunto de todas as chaves que começam com a mesma sequência de i dígitos ou caracteres definidos. Na figura 1 temos um exemplo de uma árvore Trie.

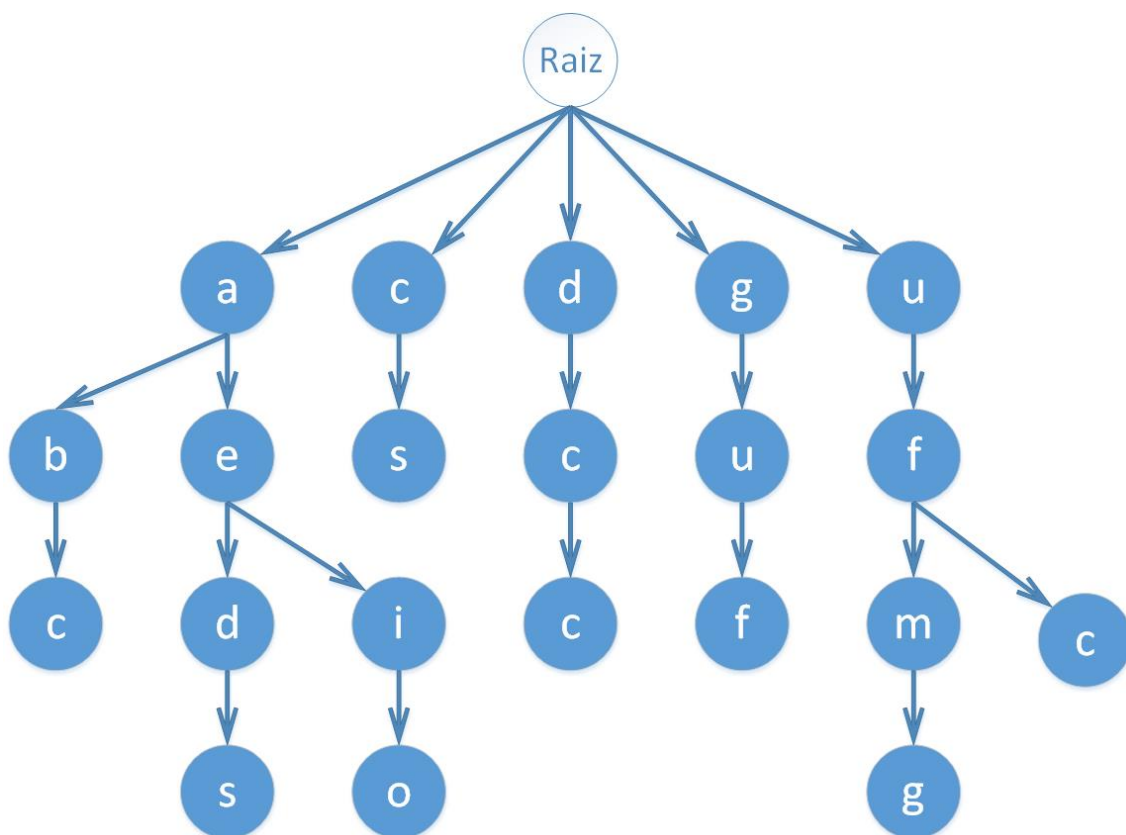


Figura 1 - Árvore Trie

## 2. Solução do Problema

A implementação foi criada a partir da definição de Tipo Abstrato de Dados (TAD), com uma estrutura de nome NO, nessa estrutura foram declaradas variáveis com o propósito de organizar o código e melhorar o desenvolvimento da estrutura de dados Trie, dentre essas variáveis temos a variável palavra do tipo bool que será verdadeira apenas no ultimo nó da palavra armazenada na Trie, facilitando a recuperação posterior das informações da estrutura. Do tipo char temos a variável letra, que receberá o caractere do nó respectivo, ocorrências e uma variável que terá valor zero para todos, menos no ultimo nó de cada palavra armazenada na árvore, essa variável irá verificar quantas vezes a palavra repetiu no texto, a última variável de nome Caracteres do tipo struct NO é um vetor de ponteiros com tamanho ALFABETO (26), que irá armazenar os caracteres subsequentes até o fim da palavra. Ainda na definição da estrutura foi definido um tipo Trie a partir do tipo NO\*, a função dessa variável é servir como um nó especial raiz, por ser um ponteiro de ponteiro isso ajuda a diferenciá-lo dos outros nós.

### 2.1 Funções e Procedimentos

As funções foram declaradas da seguinte forma no Tipo Abstrato de Dados:

**Trie\* cria\_Trie();** : Essa função tem como objetivo criar e inicializar um nó especial do tipo Trie, esse nó será a raiz da árvore, começamos a função alocando espaço para uma variável raiz do tipo Trie\*, como raiz é um ponteiro de ponteiro vamos alocar também espaço para uma outra na variável no tipo NO\* inicializá-la, e apontar o conteúdo da raiz para esse novo nó, faremos isso a criação e inicialização do novo nó usando a função cria\_No(char c) que será descrita logo abaixo, com isso já podemos retornar um nó raiz devidamente inicializado.

**NO\* cria\_NO(char c);** : A função cria nó que já foi utilizada na função cria\_Trie() irá alocar espaço para um novo nó e inicializá-lo, a alocação é feita usando a função malloc e todos os vinte e seis nós internos são inicializados com NULL utilizando um loop iterando de zero até o tamanho do alfabeto.

**int indice\_Trie(char c);** : Função simples definida para retornar um inteiro entre zero e vinte e seis a partir de um caractere, esse inteiro será utilizado como um índice para visitar o campo Caractere dos nós.

**int insere\_Trie(Trie\* raiz, char\* texto);** : Essa função é uma das mais importantes da estrutura de dados proposta, pois é ela que irá inserir os nós na árvore Trie. A inserção é feita iterativamente utilizando dois laços while, primeiramente ela recebe dois parâmetros, um do tipo Trie\*, que é a raiz da árvore e outro do tipo char\* que é a palavra a ser inserida na árvore. É definido um nó atual que irá auxiliar o caminhamento na árvore sem a perda do apontador principal raiz, também é definida uma variável que armazena o tamanho da palavra; dois laços while são utilizados para percorrer a árvore: o primeiro itera enquanto o nó do respectivo caractere a ser inserido existir e a variável tamanho do texto for maior do que zero, enquanto satisfeita essas condições descemos na árvore, vamos ao próximo caractere da palavra a ser inserida e diminuimos o tamanho da palavra. Um caso especial foi definido: se percorremos toda a palavra e todos os nós referentes aos seus caractere já existirem, então o último nó visitado será o fim da palavra, essa condição foi colocada para satisfazer os casos em que a palavra que se deseja inserir é o prefixo de uma palavra já inserida. O segundo laço é utilizado nos casos em que os nós que da palavra que se deseja inserir ainda não foram inseridos na árvore, por fim se tudo ocorreu bem é retornado um verdadeiro.

**int busca\_Trie(Trie\* raiz, char\* texto);** : A função de busca\_Trie() recebe os mesmos parâmetros da função insere\_Trie(). Também temos quase o mesmo funcionamento são declaradas as mesmas variáveis inicialmente, a diferença é que usamos apenas um laço while que irá iterar até a letra da palavra que se está buscando na árvore não existir, então se isso ocorrer retornamos falso para a busca e encerramos a função, caso a condição do laço for satisfeita até a última letra da palavra incrementamos o número de ocorrências da palavra e retornamos verdadeiro, isso é, a palavra existe na árvore.

**int ocorrencias\_Trie(Trie\* raiz, char\* texto);** : Análoga a função busca\_Trie(), a diferença é que a função de busca conta o número de ocorrências e essa função mostra na tela o número de ocorrências, essa função imprime o número de ocorrências de todas as palavras inseridas na árvore mesmo que a palavra não exista no texto onde foi feita a busca.

**void libera\_Trie(Trie \*raiz);** : Outra função muito importante para o perfeito funcionamento da estrutura de dados Trie, é a função libera\_Trie(), que irá liberar a memória de todos os nós alocados pela função insere\_Trie(), diferentemente da função insere essa função não é iterativa mas sim recursiva, a decisão de usar recursividade nessa função foi natural pois o problema de liberar os nós é um problema recursivo por natureza, mas essa função recebe um tipo Trie que é um ponteiro de ponteiro para um nó então é necessário o auxílio de outra função para liberar os nós e por fim fazer a liberação da raiz, a função libera\_No é descrita logo abaixo.

**void libera\_No(NO \*no);** : Liberar os nós é uma função recursiva mas utiliza um laço for para percorrer todo o alfabeto utilizado para construir as palavras da árvore Trie, o funcionamento é recursivo então ela percorre a árvore e vai empilhando os nós até encontrar um nó folha, depois volta liberando a pilha de nós, isso se repete para todo o alfabeto, de tamanho vinte e seis para esse problema em específico.

## 2.2 Programa Principal

No programa principal são declaradas algumas variáveis importantes, uma delas, que será utilizada ao longo de toda a execução, é a raiz do tipo Trie\*. É chamada a função cria\_Trie() para criar e inicializá-la; temos também duas variáveis do tipo inteiro: uma receberá o número de palavras do dicionário que serão inseridas na árvore Trie, e a outra irá armazenar o número de palavras do texto onde será feita a busca das palavras do dicionário. Com esses valores da quantidade de palavras do dicionário e o texto é alocado espaço para receber as palavras, um loop iterando de zero até o tamanho do alfabeto faz a leitura das palavras do dicionário e já as insere na árvore, após o término da leitura e inserção das palavras do dicionário em outro loop é feita a leitura das palavras do texto e também a verificação se elas estão inseridas na árvore essa verificação é feita pela função busca\_Trie(), ainda nesse loop já é feita a liberação dos nós que foram lidos e testados na função busca.

Após a leitura de todas as palavras do dicionário e do texto, é feita a impressão do número de ocorrências das palavras do dicionário, no mesmo loop que imprime as ocorrências é liberada a memória das palavras do dicionário.

Por fim é chamada a função libera\_Trie(), que irá liberar todos os nós inseridos na Trie finalizando a execução do programa.

## 2.3 Organização do Código, Decisões de Implementação e Detalhes Técnicos

O código está dividido em três arquivos principais: TP0.c e TP0.h implementam o Tipo Abstrato de Dados enquanto o arquivo o arquivo main.c implementa o programa principal.

O compilador utilizado foi o GCC 4.8.1 no Sistema operacional UBUNTU 16.04.

### 3. Análise de Complexidade

**Trie\* cria\_Trie();** : Na função `cria_Trie()`, primeiro alocamos uma variável raiz a custo  $O(1)$ , é chamada uma função para criar um nó a complexidade dessa função será descrito logo abaixo. O espaço ocupado pela variável raiz é 4 bytes e o tamanho do conteúdo preenchido na variável raiz pela função `cria_No()` é 112 bytes, o espaço necessário total para essa função é 116 bytes.

A complexidade de tempo é  $O(1)$  pois só temos atribuições e o espaço também será  $O(1)$  sendo que essa função necessita de 116 bytes.

**NO\* cria\_NO(char c);** : Essa função é chamada na `cria_trie()` e na função de inserir, ela tem atribuições  $O(1)$ , e um loop que itera de zero até o tamanho do alfabeto, nesse loop é alocado memória para todos os Caracteres do alfabeto a um custo  $O(\Sigma)$  onde  $\Sigma$  é o tamanho do alfabeto. A estrutura nó tem tamanho de 116 bytes então esse é o espaço necessário por essa função, com isso podemos afirmar que sua complexidade de tempo é  $O(1)$  e a complexidade de espaço é  $O(\Sigma)$ .

**int indice\_Trie(char c);** : Por ser muito simples e fazer apenas um casting ela tem complexidade de espaço é tempo  $O(1)$ , isso ocorre por ter apenas uma atribuição.

**int insere\_Trie(Trie\* raiz, char\* texto);** : A inserção dos caracteres terá o custo de tempo associado ao tamanho da palavra a ser inserida, os dois loops nunca irão juntos rodando mais vezes do que o tamanho da palavra e dentro deles temos apenas atribuições e testes de condições, a um custo  $O(1)$ . As palavras que serão inseridas formam o dicionário a função terá um custo  $O(n)$ , para o pior caso, onde  $n$ , nesse caso, é a quantidade de caracteres do dicionário menos os caracteres de espaço, já a complexidade de espaço dessa função é um pouco mais difícil de calcular, o melhor ocorrerá quando a palavra a ser inserida for prefixo de uma palavra já inserida, o pior caso ocorrerá quando todas as palavras inseridas não terem prefixo em comum entre elas, nesse caso o custo de espaço será  $O(n)$ , para o pior caso, onde  $n$  é número é quantidade de caracteres do dicionário menos os caracteres de espaço.

**int busca\_Trie(Trie\* raiz, char\* texto);** : O melhor caso dessa função ocorre quando a palavra que se deseja buscar não foi inserida  $O(1)$ , o pior caso é o oposto, e ocorre quando todas as palavras que deseja buscar foram inseridas na árvore Trie, então a complexidade de tempo é  $O(N)$ , onde  $N$  é a quantidade de caracteres do texto em que se está efetuando menos os caracteres de espaço, nessa função a complexidade de espaço é  $O(1)$  para qualquer caso.

**int ocorrencias\_Trie(Trie\* raiz, char\* texto);** : Função muito parecida com busca a diferença que ele itera em cima das palavras do dicionário, terá um custo de tempo igual para o melhor e o pior caso, pois todas as palavras estão na árvore trie com isso a complexidade de tempo será  $O(n)$ , para qualquer caso, onde  $n$ , nesse caso, é a quantidade de caracteres do dicionário menos os caracteres de espaço, a complexidade de espaço será  $O(1)$ , não temos que alocar nada, ocorreu apenas um caminhamento na árvore.

**void libera\_Trie(Trie\* raiz);** : Essa função é associada a função abaixo, pois chamamos a função `libera_No()`, essa função terá a complexidade associada a função abaixo, como além dessa chamada temos apenas a chamada da função `free()`, podemos dizer que sua complexidade é a mesma da função `libera_No()`.

**void libera\_No(NO\* no);** : A função para liberar foi definida recursivamente, e a sua complexidade de tempo e espaço será a mesma para o pior caso e o melhor caso,  $O(n)$ ,

onde n é o número de nós existentes na árvore, temos que percorrer todos os nós e empilhar até chegar no nó folha.

#### 4. Avaliação Experimental

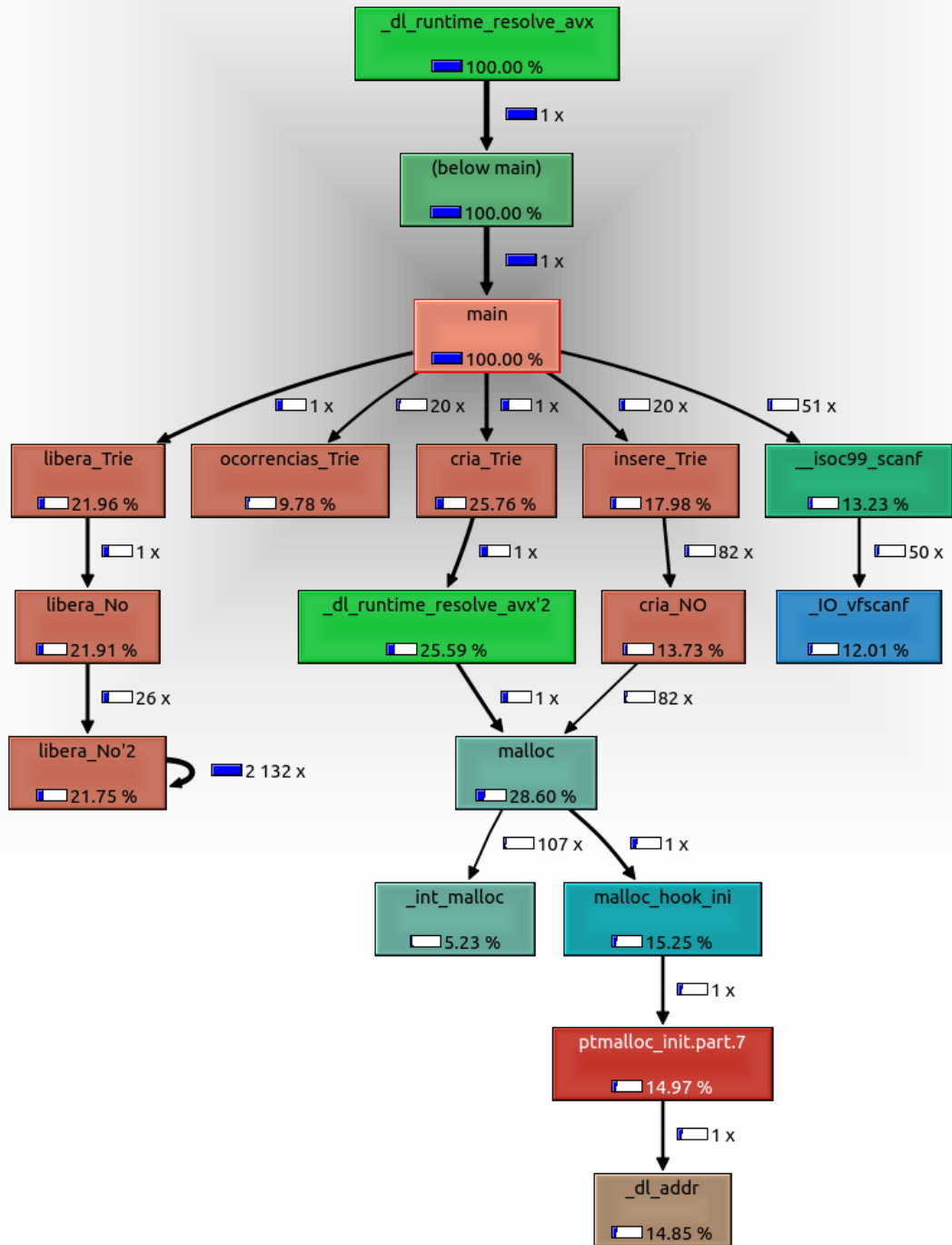
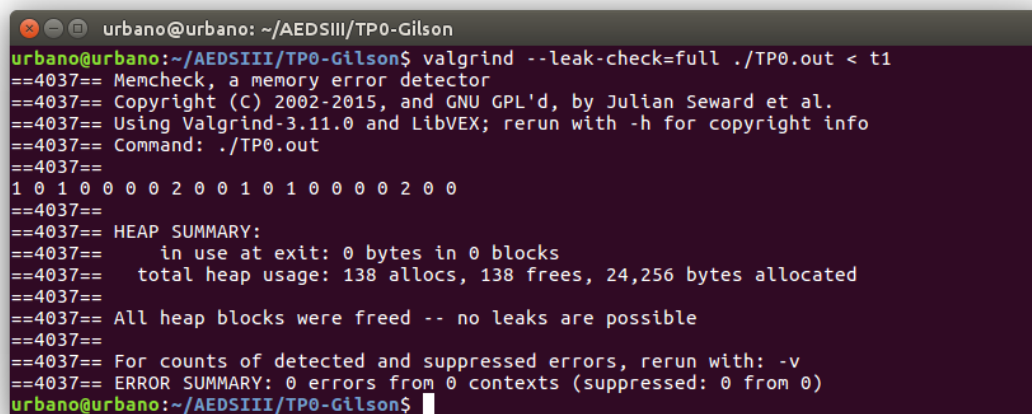


Figura 2 - Fluxo de Execução do Programa

A figura 2 foi criada usando uma flag do callgrind no valgrind, ela mostra o fluxo de execução do programa para a entrada de teste TOY 1, podemos observar que a função `insere_Trie()` é chamada vinte vezes pois o dicionário contém 20 palavras, observamos também a chamada recursiva da função `libera_No()`, esse fluxograma também traz informações sobre o percentual de quanto cada função gasta na quantidade total de execução do programa.

#### 4.1 Testes

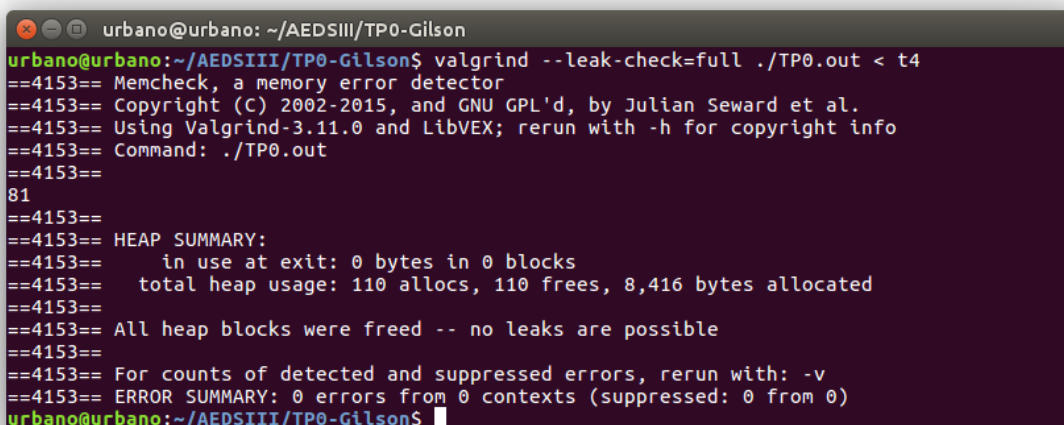
Vários testes foram realizados com o programa de forma a verificar o seu perfeito funcionamento. Os testes foram realizados em Intel Core i5, 6Gb de memória com um Intel Core i5-4210U rodando a 1.70GHz, e sistema operacional Ubuntu 16.04 LTS.

A terminal window with a dark background and light text. The prompt is 'urbano@urbano: ~/AEDSIII/TP0-Gilson'. The command executed is 'valgrind --leak-check=full ./TP0.out < t1'. The output shows Valgrind version 3.11.0, copyright information, and a heap summary. The heap summary indicates 138 allocations, 138 frees, and 24,256 bytes allocated. It also states that all heap blocks were freed and no leaks are possible.

```
urbano@urbano: ~/AEDSIII/TP0-Gilson$ valgrind --leak-check=full ./TP0.out < t1
==4037== Memcheck, a memory error detector
==4037== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4037== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4037== Command: ./TP0.out
==4037==
1 0 1 0 0 0 0 2 0 0 1 0 1 0 0 0 2 0 0
==4037==
==4037== HEAP SUMMARY:
==4037==   in use at exit: 0 bytes in 0 blocks
==4037==   total heap usage: 138 allocs, 138 frees, 24,256 bytes allocated
==4037==
==4037== All heap blocks were freed -- no leaks are possible
==4037==
==4037== For counts of detected and suppressed errors, rerun with: -v
==4037== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
urbano@urbano:~/AEDSIII/TP0-Gilson$
```

Figura 3 - Teste de Memória (TOY 1)

A partir das entradas de testes fornecidas e também das entradas criadas, foram efetuados diversos testes a fim de varrer a maior quantidade de casos possíveis, pudemos verificar alguns detalhes, um dos principais foi sobre a memória, os testes verificaram a possibilidade do programa vazar memória ele obteve êxito em todos os testes nesse aspecto, algumas demonstrações podem ser visualizadas nas Figuras 3 e 4.

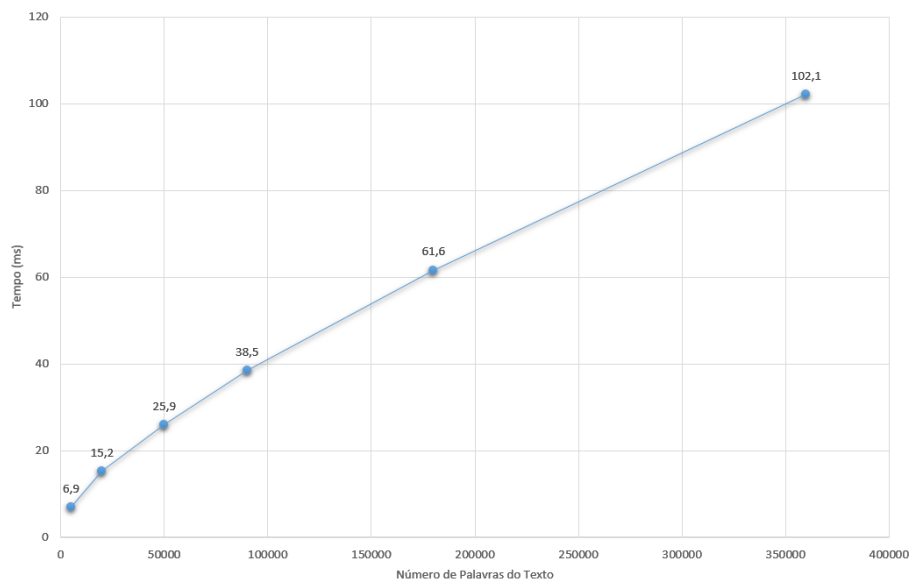
A terminal window with a dark background and light text. The prompt is 'urbano@urbano: ~/AEDSIII/TP0-Gilson'. The command executed is 'valgrind --leak-check=full ./TP0.out < t4'. The output shows Valgrind version 3.11.0, copyright information, and a heap summary. The heap summary indicates 110 allocations, 110 frees, and 8,416 bytes allocated. It also states that all heap blocks were freed and no leaks are possible.

```
urbano@urbano: ~/AEDSIII/TP0-Gilson$ valgrind --leak-check=full ./TP0.out < t4
==4153== Memcheck, a memory error detector
==4153== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4153== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4153== Command: ./TP0.out
==4153==
81
==4153==
==4153== HEAP SUMMARY:
==4153==   in use at exit: 0 bytes in 0 blocks
==4153==   total heap usage: 110 allocs, 110 frees, 8,416 bytes allocated
==4153==
==4153== All heap blocks were freed -- no leaks are possible
==4153==
==4153== For counts of detected and suppressed errors, rerun with: -v
==4153== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
urbano@urbano:~/AEDSIII/TP0-Gilson$
```

Figura 4 - Teste de Memória (TOY 4)

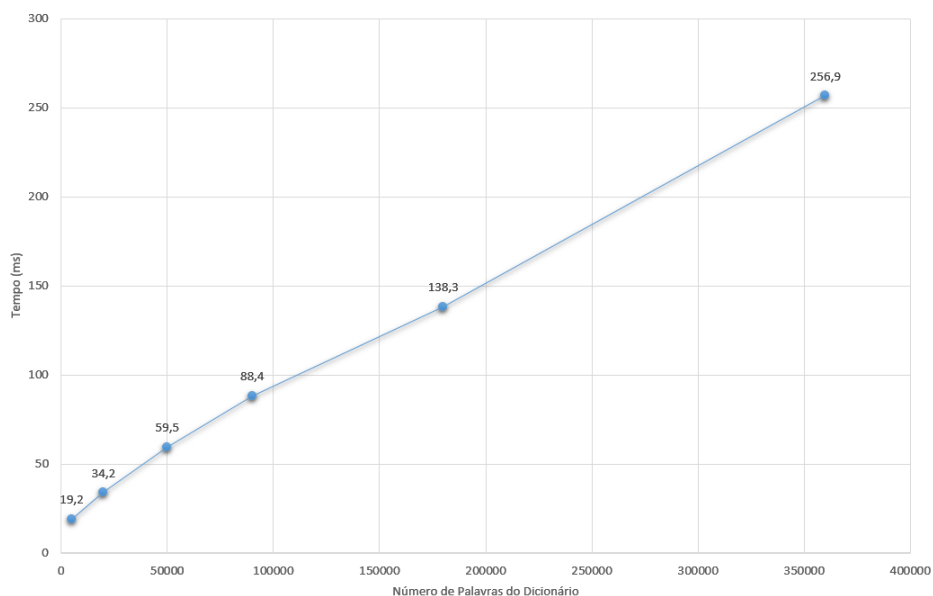
Foi utilizado o Valgrind com a flag `-leak-check=full` para verificar possíveis vazamentos de memória e se toda a memória alocada foi liberada de forma correta, o que também podemos verificar na figura 3 e 4 que ocorreu de forma correta, mas não apenas nessas entradas a verificação dessa afirmação foi testada em diversas outras entradas para chegarmos a essa conclusão.

A partir dos dados do Valgrind e do recurso Time do terminal do Linux foram criados diversos gráficos para demonstrar a complexidade de tempo e espaço do programa, logo abaixo podemos conferir alguns gráficos e seus detalhes.



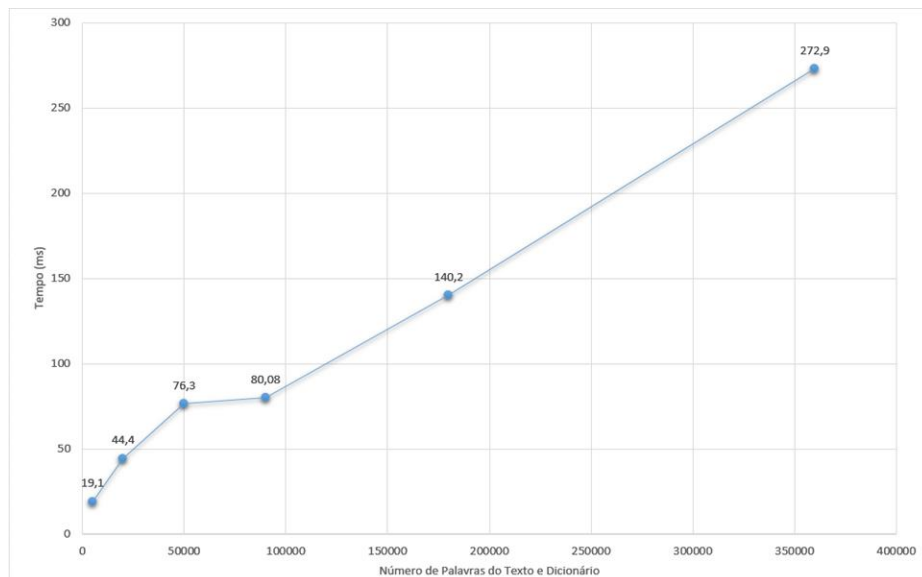
*Figura 5 - Número de Palavras do Dicionário Fixado*

A figura 5 mostra um gráfico de tempo associado ao número de palavras do texto, nesse caso o número de palavras do dicionário foi fixado em quinhentas palavras e aumentamos o número de palavra do texto, o resultado como pode ser observado é um gráfico crescente de maneira logarítmica, para se obter os valores de tempo foi feita uma média simples da mesma entrada, rodando ela por dez vezes no mesmo computador sequencialmente, essa média também será utilizada nos gráficos a seguir.



*Figura 6 - Número de Palavras do Texto Fixado*

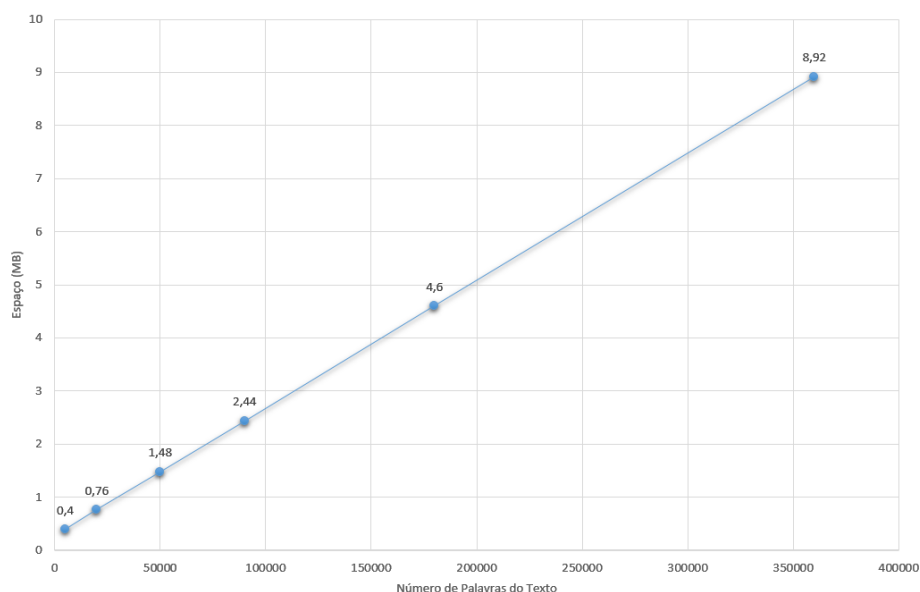
Na figura 6, foi feita o contrário o número de palavras do texto foi ficado em quinhentos e o número de palavras do dicionário foi a variável, o resultado gráfico é parecido, o que diferiu muito os dois gráficos foi a escala de tempo, quando o número de palavras do dicionário se tornou a variável o tempo mais do que dobrou, isso significa que é menos custoso fazer a busca no dicionário (Trie) do que inserir as palavras na árvore Trie, um último gráfico de tempo foi gerado no caso do número de palavras do texto e do dicionário forem iguais. Podemos conferi-lo na figura 7.



*Figura 7 - Número de Palavras do Texto e Dicionário iguais*

Temos um início logarítmico e depois o gráfico cresce linearmente isso pode ter ocorrido pelo motivo que as palavras foram copiadas e coladas várias vezes até chegar no número de trezentas e noventa mil palavras, isso faz com que as palavras se repitam muito, a busca é linear então logo após a inserção que seria o início do gráfico temos observado essa faixa linear no gráfico.

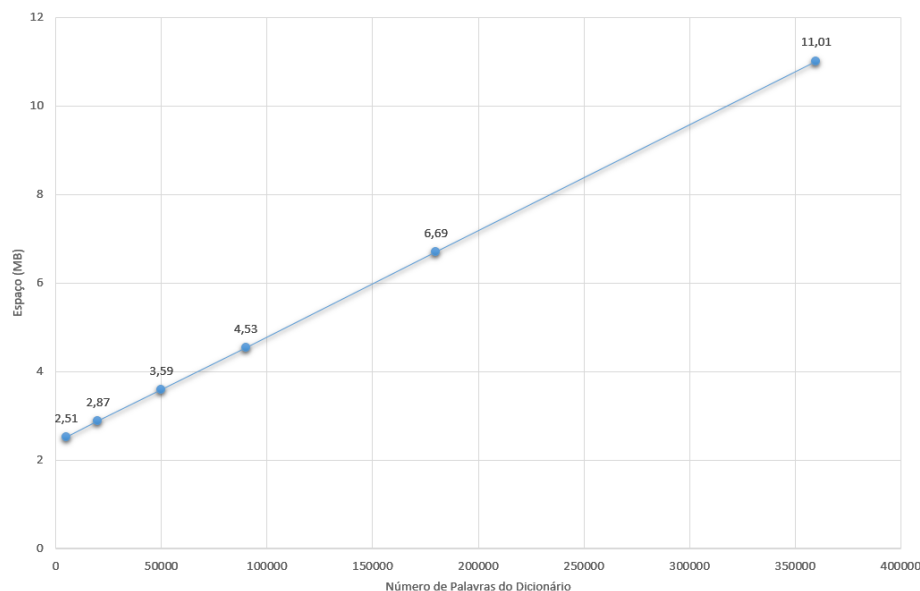
Depois dos gráficos de tempo e análises que foram efetuados para complexidade de tempo, também efetuamos para os gráficos sobre a quantidade de espaço que o programa utiliza para algumas entradas.



*Figura 8 - Número de Palavras do Dicionário Fixado*



O gráfico da figura 8 cresce de forma linear, nesse primeiro caso o número de palavras do dicionário foi fixado em quinhentas palavras e houve a variação na quantidade de palavras do texto, no segundo caso que pode ser visualizado na figura 9 logo abaixo foi efetuado o caso contrário quando o número de palavras do texto fixado em quinhentas palavras e o número de palavras do texto foi a variável; Podemos observar que quando o número de palavras foi a variável consumo de memória foi maior mas ainda seguiu de forma linear assim como no caso onde o número de palavras do texto foi a variável.



*Figura 9 - Figura 8 - Número de Palavras do Texto Fixado*

## 5. Conclusão

A implementação ocorreu de forma tranquila, os algoritmos já haviam sido estudados em Algoritmos e Estrutura de Dados II, foi necessário algumas pesquisas e revisões sobre o assunto a fim de lembrar alguns detalhes da estrutura de dados, mas por fim pudemos concluir o trabalho, com o resultado ficando dentro do esperado.

Sendo que a maior dificuldade foi na adaptação aos requisitos da disciplina, e também na busca por entradas de tamanhos grandes para tentar prever o maior número de casos diferentes, o computador chegou a travar muitas vezes ao tentar editar um bloco de texto com mais de quatrocentas mil palavras, mas com um pouco de calma e leitura pude entender os detalhes necessários e requisitos a serem tratados no problema.

Por fim o trabalho pode ser solucionado, e alguns detalhes devem ressaltados; mesmo não sendo o melhor algoritmo para pesquisa digital a estrutura de dados Trie, obteve ótimos resultados para esse problema, por sem uma árvore de prefixos ela ganha muito em velocidade de execução e diminuição da complexidade de tempo e espaço, isso se comparado a outros métodos menos inteligentes, além disso esse trabalho aguça a curiosidade e faz com que possamos ver de outra forma a área de pesquisa digital, abrindo a possibilidade de pesquisa e trabalhos futuros nessa área.

Com tudo isso problema da frequência de palavras caiu muito bem para essa estrutura de dados (árvore Trie), podendo ser solucionado cumprindo todos os requisitos e parâmetros mencionado na descrição do problema.

## 6. Referências

- [1] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 4ª Edição, Editora Pionera, 1999
- [2] André Backes, Linguagem C: Completa e Descomplicada, 1ª Edição, Elsevier, 2012
- [3] Kyle Loudon, Mastering Algorithms with C, 1ª Edição, Editora O'Reilly
- [4] Dennis M. Ritchi, C Programming Language, 2ª Edição
- [5] Robert Sedwick, Algorithms, Algorithms in C, 1990, ADDISON-WESLEY
- [6] Thomas H. Cormen ..., Algoritmos - Teoria e Prática, 2ª Edição, Elsevier, 2002
- [7] Jon Kleinberg, Eva Tardos, Algorithm design, 1ª Edição, Pearson Education, 2006
- [8] Jayme Luiz Szwarcfiter e Lilian Markenzon, Estruturas de Dados e Seus Algoritmos, 1994, LTC