Trabalho 00
Algorítimos e Estruturas de Dados III

Anagrama

Engenharia de Sistemas
Paulo Cirino Ribeiro Neto
2012022345

1 Definição do Problema

Segundo o Wikipedia[1], “Um anagrama (...) é uma espécie de jogo de palavras, resultando do rearranjo das letras de uma palavra ou frase para produzir outras palavras, utilizando todas as letras originais exatamente uma vez.” .

Assim a resolução desse trabalho está relacionada a leitura de listas de palavras a partir da entrada padrão *stdin*, e à análise dessas palavras de forma que o programa feito deve contar quantas vezes cada conjunto de rearranjos para cada lista foram inseridos na entrada. Após a contagem desses números o programa deve também os imprimir de forma ordenada decrescente.

2 Solução do Problema

Para que o programa desenvolvido pudesse realizar tal tarefa, utilizamos a idéia presente no pseudo-código abaixo:

```
begin:
TipoLista Lista;
TipoPalavra Palavra;
while( ExistePalavra() == TRUE)
{
        Palavra = LePalavra(stdin);
        OrdenaPalavra(Palavra);
        if( PalavraNaLista(Lista, Palavra) )
        {
                Lista[Palavra]->NumDeRepetições++;
        }
        else
        {
                InserePalavraNaLista(Lista, Palavra);
        }

        Imprime (  Ordena(Lista->NumDeRepetições) );
}
End;
```

A idéa presente nesse pseudo-código é a de que o programa deve lever uma lista de palavras enquanto existir palavras, então o programa deve ler e guardar essa palavra, depois ess palavra é ordenada de forma alfabética para que seja mais facil saber se ela está presente na lista. Caso ela não esteja, ela é inserida, caso ela já esteja presente existe um contador de quantas vez esssa palavra já foi repetida em todas a posições da lista que é incrementado.

3 Implementação

3.1 Bibliotécas Utilizadas

Para a implementação desse trabalho utilizamos as bibliotecas *stdio.h*, *stdlib.h*, e *string.h* . As funções utilizadas foram:

stdio.h :

- **scanf()** : Para Leitura da characteres
- **printf()** : Para impressão dos resultados

stdlib.h

- **malloc()** e **realloc()**: Para alocação dinâmica das palavras lidas e do vetor com o número de repetições de cada arranjo de palavra.
- **qsort()** : Para ordenação alfabética das palavras e para a ordenação do vetor com o número de repetições de cada arranjo de palavra.
- **free()** : Para liberar todos os ponteiros das listas e palavras do programa.

String.h

- **strcpy()**: Para copiar palavra de uma estrutura **char *** auxiliar do programa para dentro da lista.
- **strcmp()**: Para comparar a palavra lida com as palavras já presentes na lista.

3.2 Implementações .h

Para axilio da implementação da main foram criados outros 3 arquvios *.h* :

3.2.1 Lista.h

Esse header define a implementação de uma estrutura tipo Lista encadeada para que as palavras lidas possam ser armazenadas e trabalhadas em memoria.

Além disso esse header também define 2 outras funções:

- **void liberaLista(Lista L) :** que serve como um metodo para a liberação de todos os ponteiros do tipo char * que são utilizados na estrutura para guardar as palavras, além é claro da liberação também de todos os node.
- **Lista novaPalavraLista(char * palavar, int sizePalavra, Lista L) :** que trabalho no sentido de inserir uma nova palavra na lista.

3.2.2 TAD.h

Esse é um *header file* que foi criado para manipular as lista de palavras, de forma mais clara a estrutura tipo TAD definida nesse arquivos e suas 5 funções foram definidas para que não fosse necessário manipular a lista diretamente de dentro da *main* .

A estrutura TAD têm um apontador para o topo de uma lista de palavras(arranjos) e um contador de quantos arranjos diferentes existem nessa lista.

As 5 funções são:

- **TAD initTAD() :** que inicializa e seta os parametros da TAD.
- **void insereTAD(TAD T, char * palavra, int sizePalvra) :** que percorre a lista de arranjos procurando a palavra *"palavra"* e quando não à encontra adiciona a palavra na lista, mas quando a palavra é encontrada a função incrmenta a contagem desse arranjo.
- **void liberaTAD(TAD T) :** função para liberar todos os ponteiros presentes nesse TAD
- **int * vetorRespota(TAD T) :** função que percorre toda a lista de arranjos e faz um vetor com a quantidade de repetições de todos os arranjos da lista, e depois retorna essa vetor de forma ordenada.
- **void printResposta(TAD T) :** função que utiliza a função ***vetorResposta*** para imprimir a resposta pedida na especificação desse trabalho.

3.2.3 Auxiliar.h

As 3 funções definidas nesse arquivo são utilizadas na main como forma de auxiliar no tratamento das palavras antes de elas serem inseridas no TAD.

- **void maiuscula(char * palavra, int sizePalavra) :** essa função têm exatamente o mesmo resultado da função *toupper* da biblioteca *ctype.h*, onde nesse caso as palavras são transformadas em maiusulas via comparação com tabela ASC.
- **int comparaInt(...) :** função para ordenar um vetor do tipo **int** com auxilio do metodo qsort da *stdlib.h* .
- **int comparaChar(...) :** função para ser utilizada em conjunto com o método *qsort* da *stdlib.h* para ordenar palavras.

4 Análise Teórica da Complexidade

A análise de complexidade das funções e métodos utilizdos nesse programa segue de acordo com as tabelas abaixo:

4.1 Análise temporal

	Pior Caso	Melhor Caso
void liberaLista	O(n)	O(n)
Lista novaPalavraLista	O(1)	O(1)
TAD initTAD	O(1)	O(1)
void InsereTad	O(n)	O(1)
int * vetorResposta	O(n)	O(n)
void liberaTad	O(n)	O(n)
void printResposta	O(n)	O(n)
int comparaInt	O(n)	O(n)
int comparaChar	O(n)	O(n)
void maiuscula	O(n)	O(n)

De forma geral, podemos perceber que o nosso algorítimo tem complexidade teórica no pior de todos os casos de $O(n^2)$ por causa da função qsort que utilizamos para organizar as palavras.

Mas quando analisamos realmente o código e a situação que é a entrada, percebemos que em um caso geral o algorítimo têm uma eficiência muito melhor que isso

Quando analisamos todas a funções que foram feitas, é nítido que as principais e maiores funções são de ordem O(n), e fica claro nos teste que o algorítimo têm realmente essa complexidade na prática.

4.2 Análise do Espaço

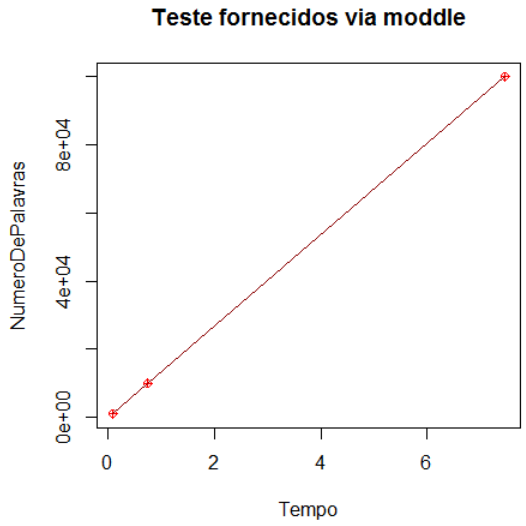
É facil perceber que o espaço que é gasto pelo programa é gasto em sua maioria para o armazenamento da lista de arranjos de palavras já inseridas. Nesse caso a complexidade é O(n), onde o espaço gasto para armazenar essas variáveis é linearmente dependente do número de arranjos de palavras diferentes.

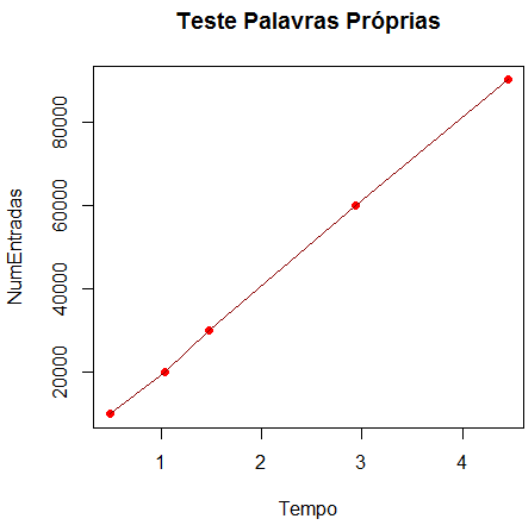
5 Análise experimental

Para a análise experimental utilizamos um computador i5-33170 1.70GHz com 5,87Gb de RAM utilizáveis em sistema Ubuntu 64-bits.

Para realização dos testes utilizamos *pipes* de forma que arquivos txt são a entrada para cada arquivo de teste diferente que foi utilizado e o tempo medido é a média de 5 iterações de cada teste. Além disso para cada teste foram utilizados 3 arquivos.

Os arquivos e as palavras utilizadas são provenientes de 3 fontes, o moodle, o site www.random.org, e o site www.unit-conversion.info .





O primeiro teste foi fornecido via moodle pela monitora Camila, nele temos que haviam 10 casos testes com 10 mil,20 mil e 100 mil palavras em cada caso, esse teste é importante para mostrar tanto a velocidade quanto o gerenciamento de memoria(ou *memory leaks*) do programa.

No segundo teste temos alguns números de entradas diferentes, nesse caso foi utilizado 1 lista para cada teste, sendo todas as palavras de cada lista unica e todas com 25 caracteres, para fazer esse teste foi utilizado um arquivo *.txt* .

Além dos testes mostrados por esses 2 gráficos, foram gerados outros testes, 1 com 50 casos de 100000 palavras cada caso para o teste de memória, e outros diversos testes menores feitos a mão para testar os resultados do programa.

Podemos perceber além do que foi dito, que os gráficos apresentam realmente um comportamento retilineo, tipo de algorítimos com complexidade O(n), assim como haviamos previsto que ocorreria.

6 Mensões Finais

6.1 Limitações do Program

Para esse programa não foram feitos nenhum tipo de programação contra erros, o problema foi resolvido da forma mais simples e direta possível que pude pensar e não me preucupei com erros fora do escopo da definição do trabalho.

Além dessa limitação o código também têm limitações quanto a proteção de estouro de memória. Uma vez que não *setei* nenhum tipo de proteção contra um número muito grande de palavras nem contra palavras muito grandes.

6.2 Conclusões

Foi concluido que este trabalho têm a inteção de refamiliarizar os alunos com a programação em C, assim durante este trabalho não foi utilizado por minha parte nenhum dos conhecimentos oferecidos durante as aulas até o momento.

De forma geral também concluí que os resultados dos experimentos práticos foram dentro do experado já que os resultados obtidos estavam corretos. Além disso o código funcionou de forma eficiente e rápida durante esses testes, e seu tempo de execução não fugiu do experado quando levamos em conta a complexidade prevista.

6.3 Considerações

Gostaria de comentar que na madrugada de quinta para sexta-feira da semana passada meu código passou em todos os testes do prático, mas infelizmente ele ainda não estava em sua versão final porque faltavam comentários e também faltava organizar as funções em arquivos .c .h fora do main .

Assim alterei meu código e gereito testes próprios mas infelizmente não pude validar a versão final no prático.