Universidade Federal de Minas Gerais

Instituto de Ciências Exatas / Departamento de Ciência da Computação

Disciplina: Algoritmos e Estruturas de Dados III

Professores: Wagner Meira Junior, Marcos Augusto Menezes Vieira
Aluno: Caíque Bruno Fortunato

Documentação do Trabalho Prático 0 – Anagramas

1. Introdução

O objetivo desse trabalho prático é agrupar grupos de anagramas em uma lista de palavras distintas com ou sem sentido lógico, retornando para o usuário o tamanho de cada grupo em ordem decrescente.

Vale recordar que anagrama é uma palavra constituída através da alteração das letras de outra palavra. *casa*, por exemplo, é um anagrama de *saca* assim como *alegria* é anagrama de *alergia*.

Cada lista de palavras constituída na entrada pode ter até 10^6 palavras, separadas por um espaço, que podem conter até no máximo 50 caracteres. É considerado também um limite de no máximo 10 listas de palavras separadas por linhas, sendo que todos os elementos das listas estão em letras minúsculas.

Através de algumas análises e estudos, a maneira mais viável encontrada para resolver esse problema foi através do uso de uma estrutura de dados Lista Dinâmica Encadeada, onde cada elemento da lista contém uma palavra diferente e um contador para a quantidade da entrada dessa palavra em suas diversas variações de anagrama.

Assim, no final da execução do programa é apresentado a quantidade de variações das diferentes palavras da entrada que, após uma ordenação, soluciona com sucesso o problema proposto.

Com a solução do problema será possível praticar conceitos de programação aprendidos em Algoritmos e Estruturas de Dados I e II, como Listas Encadeadas, Structs, ponteiros,

manipulação de strings e diversas outras ferramentas presentes nas bibliotecas da linguagem C.

Além disso, o agrupamento de anagramas pode ser importante em Bioinformática, sendo possível saber, por exemplo quais sequencias de DNA têm a mesma quantidade de bases, entre outras aplicações úteis no dia-a-dia de alguns profissionais e usuários.

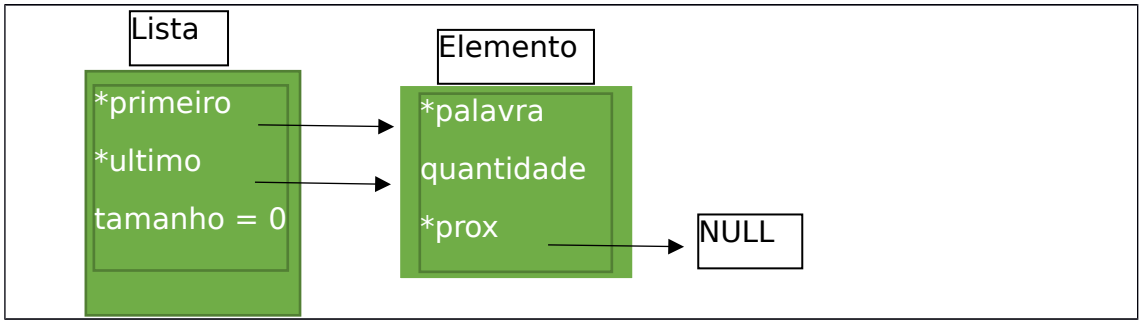
2. Modelagem do problema

A entrada é composta por no máximo 10 listas de palavras de tamanhos diversos não especificados, assim, para não haver desperdício de memória, é usado uma lista encadeada alocada dinamicamente que aumenta de tamanho a cada palavra diferente que não seja anagrama de outra contida na lista. A estrutura de dados é utilizada a cada nova lista de palavras, ou seja, se tivermos três listas de palavras na entrada, a Lista Encadeada será alocada três vezes, retornando o resultado antes de cada nova interação.

Supondo que a lista tem tamanho um e os seguintes elementos, conforme o primeiro caso de teste apresentado na especificação:

gato cama aed gota saed toga cama

É então criada e iniciada uma lista dinâmica encadeada que contém dois apontadores: um para o primeiro elemento e outro para o último, além do tamanho, que evidentemente se inicia em 0. Além disso, é alocado espaço para uma “célula cabeça”, um tipo Elemento que contém um vetor de palavras, a quantidade e um apontador para o próximo elemento, que aponta para NULL. Só então, a partir dessa célula cabeça que será inserido os próximos elementos.

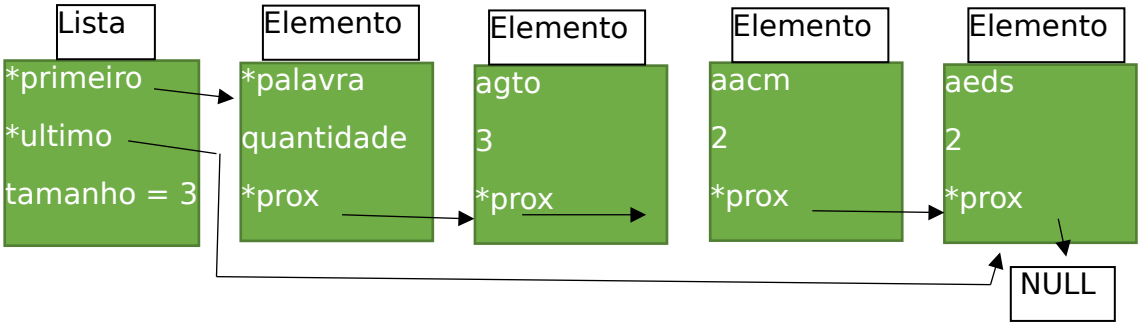


Assim, como é apenas uma lista de exemplo, usaremos apenas uma vez a estrutura. Após a criação da lista, começaremos então ler cada palavra presente na lista até que seja encontrado um ‘\n’, ou seja, uma quebra de linha. Para cada palavra lida, é guardada seu tamanho e a também ordenamos, *isso porque cada anagrama tem os mesmos caracteres e o mesmo tamanho*. Como no exemplo já citado, casa e saca, quando ordenados são a mesma palavra: aacs e aacs, ou seja, compõem um anagrama.

Mas, antes de inserir a palavra na lista, alguns cuidados são tomados: primeiro é feita uma pesquisa na lista, se a palavra existe ela não é inserida, apenas incrementa seu tamanho, mas se não existe ela é inserida e seu tamanho inicia em 1. Assim, no final teremos todas as palavras diferentes e a quantidade de anagramas.

Após a execução do programa no caso de teste apresentado acima, a lista final seria da seguinte maneira:

Leitura: gato (ordenado = agot), como não existe na lista, insere.
cama (ordenado = aacm), como não existe na lista, insere.
aeds (odenado = aeds), como não existe na lista, insere.
gota (odenado = agot), como existe na lista, incrementa 1.
saed (ordenado = aeds), como existe na lista, incrementa 1.
toga (ordenado = agot), como existe na lista, incrementa 1.
maca (odenado = aacm), como existe na lista, incrementa 1.



Após o término da entrada da lista de palavras, a lista é percorrida desde o primeiro elemento até o último e todas as quantidades são copiadas em um vetor auxiliar que possui o tamanho da lista. O vetor é ordenado em ordem decrescente e exibido como forma do resultado, resolvendo o problema proposto.

3. Análise de complexidade

Aqui será feita uma análise de complexidade das funções do programa, com uma justificativa para tal escolha.

3.1 Complexidade de tempo

Função compara_numeros e compara_caractere: Tem complexidade O(1) já que nessa função há apenas uma condição de comparação entre dois caracteres/números, sendo executada em um número fixo de vezes, no caso uma vez.

Função *cria_lista: Tem complexidade O(1) já que apenas cria um elemento do tipo Lista e Elemento, trabalhando com os apontadores ao fazer a ligação da Lista com a célula cabeça. Assim, possui complexidade constante e é executada em um número fixo de vezes.

Função *pesquisa_lista: Tem complexidade O(n) já que tem complexidade linear ao percorrer a lista em busca do que é procurado, realizando um trabalho em cada elemento da lista. No pior caso, por exemplo, a função percorre todo o tamanho da lista em busca do que está procurando. Além disso, a saída produzida é um ponteiro.

Função insere_lista: Inicialmente tem complexidade O(1), já que o custo para inserir apenas um elemento na lista é constante, sendo executada apenas na hora da inserção. O custo para incrementar um se o elemento já estiver na lista também é constante, O(1). Contudo, nessa função é realizada uma busca no vetor através da função **pesquisa_lista*, que, como apresentado, contém complexidade O(n). Assim, a complexidade da função é O(n).

Função libera_lista: Tem como complexidade O(n), já que ele da free em cada elemento de uma lista de tamanho n. Assim, a complexidade é linear ao percorrer toda a lista ao liberar todos os elementos.

Função exibe_resultado: Inicialmente tem como complexidade O(n) já que ele percorre a lista de n elementos apresentando o resultado desejado em cada posição de um vetor do tamanho da lista. Contudo, nessa função é chamada uma função de ordenação, a qsort, que está presente na biblioteca stdlib.h da linguagem C e tem complexidade definida como O(n log n) + O(1) da função compara, que resulta em O(n log n). Além disso, o vetor é imprimido, exibindo todos os seus n elementos, tendo complexidade linear O(n). Assim, O(n) + O(n) + O(n log n) = O(n log n), já que sua a complexidade é regida pela função de maior custo computacional.

Função principal: Chama uma vez as funções: cria_lista, qsort, exibe_resultado e libera lista, que, como apresentado acima, possui as seguintes complexidades

respectivamente: O(1) + O(n log n) + O(n log n) + O(n). Contudo, a função insere_lista é chamada a cada nova inserção e a cada inserção ela tem que percorrer toda a lista, assim como sua complexidade é O(n), ela passa a ser $O(n^2)$ devido as n vezes que ela é chamada. Somando com as complexidades anteriores e alguns comandos constantes e lineares: *for* sendo O(n) devido ao número de entradas das listas de palavras, que é linear por processar elementos da entrada. *While* sendo O(n) pelo mesmo motivo anterior, já que processa a quantidade de palavras contidas em uma lista. *If* sendo O(1) por fazer apenas uma comparação e comando. No final, complexidade é regida pela função de maior custo computacional, ou seja, $O(n^2)$.

3.2 Complexidade espacial

A complexidade espacial ao final da execução do programa é O(n), isso porque a cada palavra lida na entrada é inserida um elemento na lista, isso se, a palavra já não for anagrama de outra já inserida, mas, no pior caso, por exemplo, se nenhuma palavra é anagrama de outra em uma entrada de n palavras, serão necessários n inserções na lista, tendo assim complexidade linear.

4. Análise Experimental

Foram realizados vários testes após a finalização do programa com casos de testes contendo entradas com os respectivos tamanhos: 10^3, 10^4, 10^5 e 10^6, que foram disponibilizados pelos monitores da disciplina de Algoritmos e Estruturas de Dados III.

Os casos de testes foram feitos em uma máquina com 3GB de memória, processador Intel Core 2 Duo x86 com sistema operacional Linux com distribuição Ubuntu 14.10 em seu compilador nativo, GCC.

Pada entrada, houveram 10 execuções diferentes e o tempo foi calculado através do comando *time ./lista ./[Nome do arquivo de teste]* através do uso da variável user, que calcula o tempo que o programa gastou para calcular as entradas disponibilizadas, ignorando o tempo demorado para que o usuário entre com os dados. Os dados das 10 execuções de cada arquivo foram guardadas para que fosse calculada a média e que o resultado final fosse obtido.

Os dados gerados foram os seguintes:

Entrada	Tempo (s)
10^3	0.0796
10^4	0.75548
10^5	7.486
10^6	75.176

Com os dados apresentados, foi desenhado um gráfico para representar a Entrada x Tempo para ficar mais claro a visualização da execução do tempo do algoritmo.

Através da análise do gráfico, foi comprovado a complexidade do arquivo, $O(n^2)$ que foi discutida anteriormente. Isso porque, a medida que a entrada cresce o tempo aumenta de maneira quadrática, algo parecido realmente com uma função $f(x) = x^2$, onde x é um número natural, se pegarmos cada vez mais casos de testes.

5. Conclusão

A implementação do trabalho ocorreu sem muitas dificuldades, através dos conhecimentos adquiridos em Algoritomos e Estruturas de Dados I e II, utilizando também funções importantes e úteis encontradas em bibliotecas presentes na linguagem C como string.h, stdio.h e stdlib.h.

Com o desenvolvimento do trabalho foi possível relembrar vários conceitos da linguagem C, a manipulação de ponteiros, listas encadeadas, entre outras ferramentas relativamente simples, mas importantes e úteis na vida de qualquer programador. Treinando tambémum pouco de lógica e melhor compreensão para a resolução do problema proposto.

Os conhecimentos adquiridos serão importantes ao decorrer da disciplina, já que estruturas como listas e suas viraçõe, pilhas e filas são necessárias para resolução de problemas em grafos, por exemplo. Além disso, a manipulação de caracteres também introduz importantes conceitos que serão estudados.

Por fim, foram aperfeiçoados diversos conceitos importantes no desenvolvimento do programa e a solução do problema ocorreu com sucesso em todos os casos de testes com complexidade $O(n^2)$, não foram encontradas muitas dificuldades no desenvolvimento do trabalho.