TRABALHO PRÁTICO 2
ANALISADOR DE TEXTO

Tarcizio Augusto Santos Lafaiete

Universidade Federal de Minas Gerais(UFMG)
Belo Horizonte - MG - Brasil

tarcizio-augusto@hotmail.com

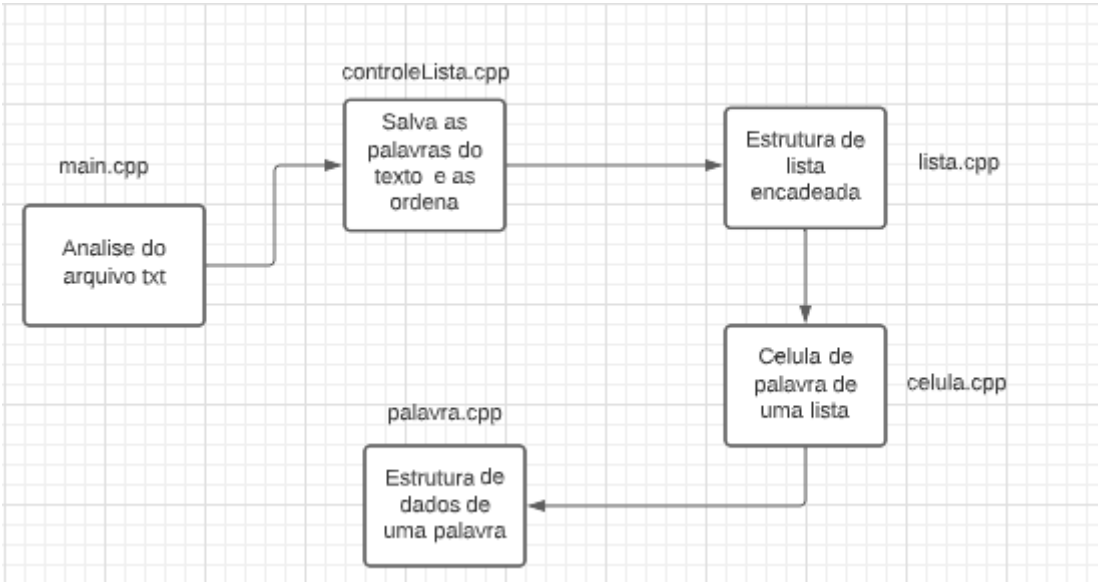
1.Introdução

O problema proposto para implementação do trabalho prático 2 foi o desenvolvimento de um programa que recebe como entrada um arquivo que possui dois blocos de texto marcados pelas tags "#ORDEM" e "#TEXTO". No bloco de texto temos um texto qualquer o qual se deseja contar as repetições das palavras e ordenar estas em ordem crescente. Já no bloco da ordem temos as 26 letras do alfabeto dispostas de forma arbitrária, esta disposição será usada para definir uma nova ordem lexicográfica a qual as palavras do texto devem ser ordenadas. Assim como resultado da análise deste arquivo geramos um novo arquivo de saída no qual as palavras e sua quantidade de repetições estão impressas em ordem crescente seguindo a ordem lexicográfica estabelecida.

Esta documentação inicia na seção 2 explicando as decisões de implementação tomadas pelo aluno e explicações do funcionamento básico das classes implementadas. Em seguida na seção 3 as informações da seção 2 são complementadas trazendo as estratégias de robustez usadas pelo desenvolvedor do programa para tratar exceções. Na seção 4 tratamos da complexidade espacial e temporal de diversos métodos implementados. Assim, após finalizar a explicação do código na seção 5 é falado sobre as análises experimentais feitas em relação ao código. Por fim, na seção 6 explica-se os processo para a compilação e execução correta do programa e a seção 7 conclui a documentação com as considerações finais do autor.

2.Implementação

Para a elaboração do analisador de texto solicitado neste trabalho prático, primeiro foi-se planejado como seria o fluxo de informações partindo do arquivo de texto com os blocos de ordem e texto até a saída com o vetor de palavras ordenados:



Tendo em mente este diagrama, além do programa principal foram criadas quatro classes,sendo elas a estrutura base de uma palavra, uma lista encadeada e suas células para guardar as palavras e uma classe de controle para montar uma lista com as palavras do texto e realizar a ordenação destas.

2.1 Estruturas de dados

Dentro da implementação do programa foi utilizada a estrutura de uma lista simplesmente encadeada para armazenar as palavras criadas pela classe palavra também implementada dentro do analisador de texto. Para a lista foram desenvolvidos os métodos padrões de inserção e remoção no final e início da lista, um método de pesquisa e um método que retorna a palavra pela sua posição na lista.

2.2 Ordenação e escolha do pivô

2.2.1 Quicksort e Seleção

Neste trabalho prático foi necessário a implementação dos métodos de ordenação Quicksort e Seleção. O Quicksort é tido como o método mais eficiente de ordenação de elementos, ele se baseia na estratégia de dividir para conquistar para ir rearranjando vetores cada vez menores para no final deixar ordenado o vetor inicial. Para sua implementação primeiro é escolhido um elemento pivô, em seguida há o particionamento do vetor de modo que os elementos à esquerda do pivô sejam menores que ele e os elementos do lado direito sejam maiores. Por fim há a chamada recursiva para ordenar os subvetores criados.

Já o método de Seleção é um dos métodos mais simples de ordenação e se baseia na ideia de selecionar no vetor o menor valor e colocá-lo na primeira posição, em seguida selecionar o segundo menor valor e colocá-lo na segunda posição e assim sucessivamente até termos o vetor completamente ordenado.

2.2.2 escolha do pivô

Na inicialização do programa é passado como argumento o número de elementos que devem ser usados para a escolha do pivô do Quicksort. A partir disso pegamos o ponto médio do vetor e selecionamos os elementos do meio do

vetor com base na quantidade de elementos passada como argumento. Com os elementos selecionados realiza-se uma ordenação destes através do método de seleção e escolhe o ponto médio destes elementos ordenados como o pivô. Caso o argumento passado seja maior que o tamanho do vetor, consideramos o elemento médio do vetor como o pivô.

2.3 Classes e main

2.3.1 Classe Palavra

Esta classe é bem simples e é utilizada para representar uma palavra, nela além de uma string com a própria palavra em si, temos um inteiro para identificar as repetições desta palavra em um texto. Nela usamos métodos comuns de get e set para a string e para o inteiro da classe, além de um método para incrementar os valor do inteiro.

2.3.2 Classe ControleLista

A classe ControleLista é usada para realizar o armazenamento das palavras do bloco de texto da entrada dentro de uma lista encadeada, além disso ele possui uma segunda lista na qual está armazenada a nova ordem lexicográfica estabelecida. Alem dos métodos de montagem das listas há um método para a ordenação das palavras seguindo a ordem definida anteriormente e impressão da ordenação em um arquivo de saída. Importante ressaltar que para o caso da lista de ordem, cada palavra corresponde a uma das letras do alfabeto e segundo a ordem pré-estabelecida no arquivo de entrada e o inteiro atribuído a cada letra corresponde a sua posição na nova ordem lexicográfica, sendo assim a primeira letra recebe o valor 65 e a última letra recebe o valor 90, esta faixa de valores foi escolhido pelo fato de ser a faixa de valores correspondente às letras de A a Z na tabela ASCII sendo assim se tornaria mais fácil fazer comparações das palavras seguindo a nova ordem com outros caracteres como números.

2.3.3 Main

Finalizando este subtópico, temos no nosso programa principal a análise dos argumentos de entrada do programa assim definindo qual o arquivo de entrada que deve ser analisado, o valor do vetor no qual o ordenação deve parar de ser feita por Quicksort e passar a ser por seleção, o valor de elementos que deve ser considerado para encontrar o pivô e o nome do arquivo de saída. Em seguida começa a análise do arquivo de entrada e a passagem das palavras para a lista de palavras do texto e a lista da ordem lexicográfica. Importante ressaltar que tanto as letras da ordem como as palavras do texto são todas passadas para o formato minúsculo para facilitar as análises requisitadas no trabalho prático, e para as palavras presentes no texto também foi feita a limpeza de elementos indesejados como “?”,”!”,”.” e “;” no final das palavras do texto.

2.4 Dados do ambiente

Este programa foi desenvolvido em C++, compilado pelo G++ da GNU Compiler Collection em computador com o processador Intel Core i3-6006 U, com 4GB de memória RAM e o Linux Mint 20.2 como sistema operacional.

3.Estratégias de Robustez

As estratégias de robustez implementadas no código foram focadas na análise dos argumentos passados no momento da execução do programa para evitar quaisquer falhas de segmentação ou outros erros originários de uma má passagem de parâmetros para o programa. A primeira medida para aumentar a robustez do programa foi tornar os valores passados pelos argumentos -m e -s como 0 por padrão, assim sendo caso um destes dois dois argumentos não seja passado no momento da execução o programa interpretará ambos como zero e seguirá a execução normal do programa. Assim, no caso do -m ele irá considerar o pivô sempre como o elemento central do vetor, já para o -s ele irá executar a ordenação inteiramente como quicksort.

Outra estratégia adotada foi para analisar se os argumentos -i e -o foram passados como parâmetro, caso um deles não tenha sido passado será exibido no terminal uma mensagem de erro avisando que este argumento não foi passado. Por fim analisando o conteúdo do arquivo de entrada, caso ele seja um arquivo vazio, ou não tenha os marcadores “#TEXTO” e “#ORDEM”, ou um destes dois blocos seja vazio ele irá retornar uma mensagem de erro sinalizando que houve uma falha na leitura do arquivo de entrada.

4.Análise de Complexidade

As análises de Complexidade do programa serão dividas em partes, sendo então analisada a complexidade de cada método das classes até chegarmos na análise de complexidade da main.

4.1 Classe Palavra

getPalavra/setPalavra - complexidade de tempo : Os métodos de get e set da palavra tem complexidade temporal O(1) já que neles são realizadas apenas operações de custo constante.

getPalavra/SetPalavra - complexidade espacial : A complexidade espacial é O(1), pois a alocação de espaço nestes métodos é constante.

getValor/SetValor - complexidade de tempo : Estes método tem complexidade O(1), uma vez que eles realizam apenas operações de custo constante.

getValor/SetValor - complexidade espacial : Tem complexidade espacial O(1), já que o espaço ocupado neste método é sempre constante.

incrementaValor - complexidade de tempo: Possui complexidade de tempo igual a O(1), pois suas operações são constantes.

incrementaValor - complexidade espacial: Sua complexidade é O(1), já que realiza alocações constantes em suas chamadas.

4.2 Classe ListaPalavra

pesquisaPalavra - complexidade de tempo: Tem a complexidade temporal O(n) no pior caso quando o elemento pesquisado está no último item lista ou não se encontra na lista e O(1) no melhor caso quando a palavra procurada está na primeira célula.

pesquisaPalavra- complexidade espacial: Possui complexidade O(1), já que este método aloca apenas uma palavra auxiliar para realizar as comparações.
inserePalavraIncio/inserePalavraFinal - complexidade de tempo: A complexidade de tempo destes métodos é O(1), uma vez que a inserção necessita apenas movimentar o ponteiro de um elemento.
inserePalavraIncio/inserePalavraFinal - complexidade espacial: A complexidade espacial é O(1), já que é alocado apenas o espaço para adicionar mais uma palavra na lista.
removePalavraInicio - complexidade de tempo: Tem complexidade temporal igual a O(1), já que a remoção é feita no início da lista, assim só precisando mover o ponteiro do primeiro para o próximo e remover a antiga primeira célula.
removePalavraInicio - complexidade espacial: Tem complexidade espacial O(1), devido a alocação de memória constante e independente da entrada.
removePalavra - complexidade de tempo: Tem complexidade temporal igual a O(n) no pior caso no qual o elemento a ser removido está na última posição ou não se encontra na lista e O(1) caso o elemento seja a primeira célula da lista.
removePalavra - complexidade espacial: Tem complexidade espacial O(1), pois a alocação de memória é constante e independente da entrada.
getPalavraPosicao - complexidade de tempo: Sua complexidade de tempo é O(n), pois o número de repetições do loop presente neste método depende do valor da posição passada como parâmetro.
getPalavraPosicao - complexidade espacial: A complexidade espacial deste método é O(1), já que apenas cria uma célula auxiliar para realizar as operações do método.
IncrementaPalavra - complexidade de tempo: Sua complexidade temporal é no pior caso O(n) quando o método precisa comparar a chave do parâmetro com todas as chaves da lista e O(1) no melhor caso, quando ele compara apenas com a primeira chave.
IncrementaPalavra - complexidade espacial: Sua complexidade espacial é O(1),uma vez que aloca espaço apenas para uma célula e palavra auxiliar.

4.3 Classe ControleLista

montaOrdem - complexidade de tempo: Este método tem complexidade temporal O(1), pois este algoritmo é possui apenas operações de custo constante.
montaOrdem - complexidade espacial: Tem complexidade espacial O(1), uma vez que aloca espaço apenas para uma nova palavra dentro da lista de ordem.
montaTexto - complexidade de tempo: A complexidade temporal deste método é O(1) quando o método pesquisaPalavra está em seu melhor caso O(1), pois neste caso todas as operações de montaTexto serão constantes. Já quando pesquisaPalavra está no seu pior caso, montaTexto será O(n), pois será a soma de O(n) de pesquisaPalavra com diversas operações O(1), ou seja, teremos O(n) + O(1)+...+O(1) = O(n).
montaTexto - complexidade espacial: Tem complexidade espacial O(1) devido a apenas alocar memória para uma palavra na lista de palavras do texto.

quicksort - complexidade de tempo: Este método tem complexidade temporal O(n log n) quando executa o algoritmo de quicksort uma vez que ele chama recursivamente o método particao que é O(n) log n vezes. Já quando executa o algoritmo de seleção ele é $O(n^2)$.
quicksort - complexidade espacial: O quicksort tem complexidade espacial O(log n), pois cada chamada recursiva do quicksort custa O(1) para a memória, porém como chamamos o quicksort log n vezes temos então O(1). log n = O(log n).
particao - complexidade de tempo: O método de partição tem complexidade O(n), uma vez que serão executadas n comparações até que os iteradores da esquerda e da direita se cruzem. Além disso podemos considerar a função escolhePivo O(1), pois o tamanho do vetor passado para esta função é constante neste caso.
particao - complexidade de espacial: Sua complexidade espacial é O(1), já que ocupa espaço constante na memória para executar as suas operações.
selecao - complexidade de tempo: O método de seleção tem uma complexidade temporal $O(n^2)$, uma vez que ele possui dois loop aninhados que iteram n vezes cada ficando então O(n).O(n) = $O(n^2)$.
selecao - complexidade espacial: O método de seleção tem complexidade espacial O(1), pois ele aloca um espaço constante na memória.
escolhePivo - complexidade de tempo: O método escolhePivo tem uma complexidade temporal $O(n^2)$, pois além de possui um loop aninhado de complexidade O(n) ele faz a chamada do método selecao, assim ficando O(n) + $O(n^2)$ = $O(n^2)$.
escolhePivo - complexidade espacial: Ele tem complexidade espacial O(n), uma vez que este método aloca na memória espaço para um vetor de n posições.
imprimiVetor - complexidade de tempo: Tem complexidade temporal O(n), já que ele é composto apenas por um loop aninhado de complexidade O(n).
imprimiVetor - complexidade espacial: Ele tem complexidade espacial O(1), pois aloca um espaço constante de memória para a sua execução.
ordena - complexidade de tempo: Este método tem complexidade temporal O(n), pois ele possui um loop aninhado de complexidade O(n log n), faz a chamada do quicksort que é O(n log n) e a chamada do imprimiVetor O(n), assim ficando O(n) + O(n log n) + O(n) = O(n log n).
ordena - complexidade espacial: Ele possui complexidade espacial O(n) pois aloca espaço para um vetor de tamanho n e aloca espaço para o quicksort que é O(log n) então temos O(n) + O(log n) = O(n) .
comparaPalavrasNovaOrdem - complexidade de tempo: Este método tem complexidade de tempo O(n) no pior caso, no qual ele precisa iterar por um palavra de n caracteres para descobrir qual é a maior. Já no melhor caso tem complexidade O(1) quando o loop é finalizado na primeira comparação.
comparaPalavrasNovaOrdem - complexidade espacial : Ele possui complexidade O(1), pois aloca um espaço constante na memória para a sua execução.

4.4 Main.cpp

main - complexidade de tempo: No programa principal temos uma complexidade O(n log n), uma vez que ele possui a chamada o parseArgs de complexidade O(1), em seguida é iniciado um loop lendo todo o arquivo de entrada que possui n palavras, tendo então complexidade O(n) e por fim chama o método ordena de complexidade O(n log n), assim temos que a complexidade do programa principal é dada por O(n) + O(1) + O(n log n) sendo então igual a O(n log n).
main - complexidade espacial: O programa principal tem complexidade espacial O(n), pois além do método ordena de complexidade O(n) chamando no programa principal todas as outras operações possuem custo de memória constante.
parseArgs - complexidade de tempo: Esta função tem complexidade temporal O(1), uma vez que o loop presente nela repete constantemente 9 vezes que é o valor passado no argumento argc dentro da função main.
parseArgs - complexidade espacial: Sua complexidade espacial é O(1), pois o espaço alocado pela função é constante e independente da entrada.
normalizaString - complexidade de tempo: A função normalizaString tem complexidade de tempo O(n), uma vez que possui um loop que percorre uma string de tamanho n.
normalizaString - complexidade espacial: Já esta função tem complexidade espacial O(1), pois ela ocupa um espaço de memória constante.
limpaAcentuacao - complexidade de tempo: Esta função tem complexidade temporal O(1), pois a função só analisa a última posição da string o que é uma operação O(1).
limpaAcentuacao - complexidade espacial: Ela possui complexidade espacial O(1), já que sua alocação de memória auxiliar é constante.

5.Análise de Experimentos

5.1 Desempenho computacional de tempo

Os experimentos realizados para medir o desempenho computacional do analisador de texto foram feitos utilizando a biblioteca de c, time.h, com ela criamos duas variáveis start e end do tipo clock_t. O clock_t é uma estrutura de dados implementada pela biblioteca time.h, no qual as variáveis deste tipo são capazes de guardar o número de sinais de clock utilizados pelo processador até o momento pelo programa. Então a start foi chamada logo antes do início do algoritmo implementado na main e a end no final do código.

Com esses valores armazenados, fazemos a diferença deles e dividimos por CLOCKS_PER_SEC que é um define da biblioteca time.h que representa o valor do número de clocks que o processador gera por segundo. Dessa divisão então conseguimos o tempo de execução do programa.

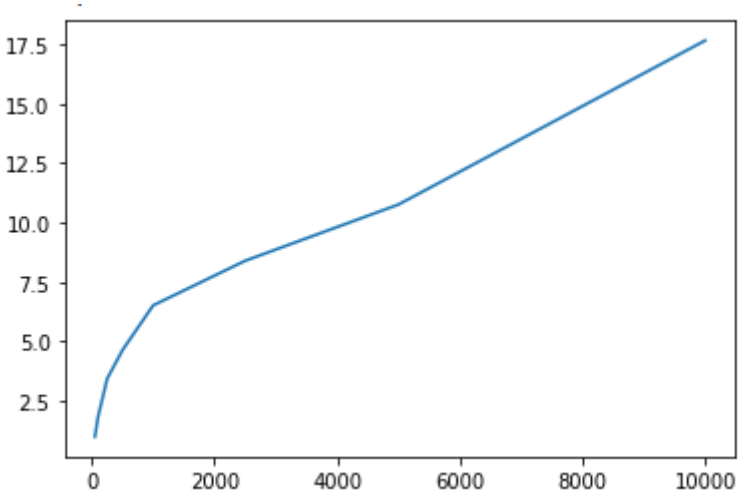
Explicado como foi feita a medição de tempo, agora pode-se detalhar como e em quais condições foram feitas as medidas. Estas medidas foram todas feitas com os argumentos -m e -s como 5 para termos um padrão entre todas as execuções

além disso o texto analisado foi gerado através do site Dummy Text Generator na qual variamos o número de palavras do texto de 100 até 10000, a partir disso tirou-se 5 medidas de cada texto.

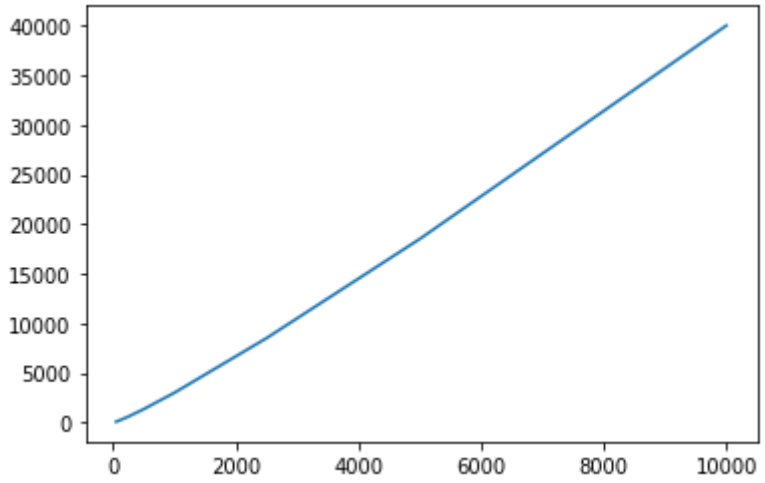
Deste experimento conseguimos a seguinte tabela:

Entrada	Medida 1	Medida 2	Medida 3	Medida 4	Medida 5	Média
50	7,26 ms	8,046 ms	5,67 ms	5,72 ms	6,165 ms	6,5722 ms
100	10,973 ms	10,397 ms	13,539 ms	13,841 ms	10,861 ms	11,9222 ms
250	19,078 ms	24,261 ms	25,272 ms	21,182 ms	22,956 ms	22,5898 ms
500	29,001 ms	27,156 ms	42,71 ms	27,14 ms	26,464 ms	30,4942 ms
1000	34,962 ms	30,063 ms	42,433 ms	48,34 ms	58,495 ms	42,8586 ms
2500	65,851 ms	43,086 ms	75,092 ms	42, 687 ms	49,211 ms	55,1854 ms
5000	62,856 ms	91,316 ms	67,836 ms	61,975 ms	69,684 ms	70,7334 ms
10000	112,815 ms	119,367 ms	125,475 ms	120,234 ms	102,398 ms	116,0578 ms

Utilizando então os dados apresentados na tabela e as funções da biblioteca matplot da linguagem python, criamos um gráfico com o tamanho das entradas como eixo x e a média de tempo obtida dividida pela média de tempo da entrada de 50 palavras como eixo y. Assim tivemos este gráfico:



Pegando agora o gráfico esperado para um programa de complexidade O(n log n) teremos o seguinte comportamento:



Nota-se que o comportamento do programa foi o esperado visto que apesar do comportamento estranho causado nas menores entradas,muito provavelmente causadas por perturbações de outras tarefas executadas pelo processador, à medida que o número de palavras no texto se tornou maior, a resposta do programa seguiu o modelo esperado.

7.Conclusão

Neste trabalho foi desenvolvido um software que analisa um arquivo de entrada, na qual contém uma definição de uma nova ordem lexicográfica e um texto que deveria ter suas palavras ordenadas através de um quicksort seguindo esta nova ordem e impressas em um arquivo de saída com o número de vezes que estas palavras repetiram no texto. Durante a execução do trabalho grandes aprendizados foram adquiridos tanto pelos erros, quanto pelos acertos durante o processo. Além do óbvio ganho de prática com o desenvolvimento de programas do zero.

Com o trabalho o autor pode adquirir mais prática e conhecimento sobre o uso dos algoritmos de ordenação. Importante ressaltar as dificuldades iniciais com o entendimento da recursividade dentro do Quicksort, uma vez que na maioria dos códigos não tem-se o costume de fazer uso de funções recursivas, além disso o processo de escolha de um pivô através da mediana de um certo grupo de valores dentro do vetor e a mudança do método de ordenação a partir de um determinado tamanho de vetor foram implementações pedidas dentro do escopo do trabalho que foram interessantes e instigaram o autor a pensar nos métodos de ordenação não somente com uma caixa fechada apresentada nos slides da disciplina, mas sim como um processo base que que contanto que não sofra alterações na base do fluxo de operações do algoritmo pode sofrer variações e melhorias com base nas necessidades do programador.

Por último também foi interessante o requisito de analisar a linha de comando do programa, já que a passagem de argumentos para o programa não foi algo tão bem exercitado em matérias anteriores.

Apêndice

Instruções de compilação

- Abra o dirétoria TP no terminal linux ou prompt comando do Windows
- Realize o comando make all
- Entre na pasta bin
- Coloque o arquivo de entrada dentro desta pasta
- e realize a linha de comando para execução do programa conforme o padrão de passagem de argumentos descrito nas especificações deste trabalho prático.