Trabalho Prático 2 da disciplina de Estrutura de Dados

Alex Eduardo Alves dos Santos
Matrícula: 2021032145

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
alexeduardoaeas@ufmg.br

1. Introdução

O problema proposto foi criar um sistema que contabiliza o número de ocorrências das palavras usadas em um texto baseado em uma nova ordem lexicográfica.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estrutura de Dados

A implementação do programa teve como base a estrutura de dados de um array de strings. Tal que o objetivo principal é ordenar esse array utilizando um método eficaz chamado de QuickSort, no entanto, ainda foi preciso tratar as entradas do arquivo de entrada. Nesse sentido, o primeiro procedimento foi realizar a leitura das informações e prepará-las para serem utilizadas nas funções posteriores. Logo, o programa se inicia realizando a leitura dos caracteres da nova ordem lexicográfica, isso foi feito ignorando os espaços que existiam entre esses caracteres e armazenando-os em um array. Foi optado por utilizar a estrutura de string invés de um array para esses dados, e foi adicionado elementos da tabela ASCII na ordem correta e os caracteres da nova ordem lexicográfica em sequência, além disso, a fim de facilitar as comparações todos os caracteres foram utilizados na sua respectiva forma minúscula.

Em seguida foi feita a leitura do texto que seria ordenado, ele se segue de maneira parecida com o anterior, primeiramente foi armazenado em uma string e depois colocado palavra por palavra em um array, com todos os caracteres na sua forma minúscula e sem as pontuações indevidas.

2.1. Algoritmo de Ordenação

Feito isto, com todos os dados prontos para serem processados, foi chamado a função do QuickSort, essa função ordena as strings, e para isso, ela conta com o auxílio de outra função chamada de ComparaStrings, que compara as strings a partir da nova ordem lexicográfica e diz a função QuickSort quais strings devem ficar antes ou depois no array.

Ainda no QuickSort, é possível notar algumas estratégias tomadas para reduzir o custo da operação de ordenar esse array de strings, o QuickSort segue um compromisso de reduzir o número de comparações e assim ser mais eficiente, para isso foi utilizado algumas estratégias como escolher um pivô, a partir da mediana de M elementos do array, M

passado como parâmetro na execução do programa, e a partir desse pivô procura-se um elemento menor que ele a direita dele no array e um menor a esquerda, ao encontrar esses elementos eles são trocados de posição no array e o programa segue realizando essa estratégia recursivamente até o vetor estar devidamente ordenado, no entanto, é possível notar que existe um ganho maior ao utilizar um método simples de ordenação como o Inserção a partir de certo momento da recursão, logo, na execução do programa um parâmetro é passado para definir em que momento esse método simples será utilizado, e quando a partição que foi sendo dividida pelo quickSort chega a um tamanho menor ou igual ao valor desse parâmetro é usado o InsertSort para ordenar o restante do array.

Finalmente, com o array ordenado é contabilizado a ocorrência das palavras no texto, para isso foi verificado quantas vezes as palavras se repetiam durante a varredura do array, e a cada vez que ela se repetia um contador é incrementado, até o momento em que a palavra não se repete mais e é impressa no arquivo de saída junto com o valor do contador.

2.2. ComparaString

Essa função é utilizada nos dois métodos de ordenação, QuickSort e InsertSort, essa função recebe duas strings e a ordem lexicográfica para classificar qual string vem antes da outra, para isso é verificado caracter por caracter das duas strings e comparado com qual carácter da ordem lexicográfica ela se identifica, ao encontrar esse caráter, o valor dele na ordem lexicográfica é atribuído a um índice da string, se elas possuírem caracteres diferentes em posições iguais, então elas se diferenciam e podem ser classificadas, se não, ou elas são completamente iguais ou parcialmente iguais, e se diferenciam pelo tamanho em que a maior tem mais caracteres que a menor não possui.

3. Análise de Complexidade

3.1 Tempo

3.1.1 Main

Começaremos a análise a partir dos procedimentos utilizados na função main, em que basicamente é feita a leitura dos dados do arquivo de entrada e são tratados para serem utilizados em funções posteriores.

O programa inicia guardando as primeiras informações dos parâmetros de execução, esses são guardados com um loop atrelado a um switch, e a complexidade é O(n) mas esperasse que esse laço seja chamado apenas 4 vezes, uma para cada parâmetro.

Em seguida é feita a leitura dos dados no arquivo de entrada, primeiramente é feita a leitura da nova ordem lexicográfica, para isso é utilizado um laço que possui complexidade O(n), N sendo o tamanho da entrada dos caracteres da ordem lexicográfica, no melhor caso em que a ordem está separada por apenas um espaço entre os caracteres, esse laço é chamado 52 vezes, mas se existirem mais espaços entre os caracteres pode aumentar linearmente, portanto, esse laço custa O(n). Após isso os caracteres são guardados novamente em uma string, que possui outros caracteres da tabela ASCII, essa operação custa O(26), uma chamada para cada caracter. Para converter todos os caracteres em seu respectivo minúsculo um laço que custa sempre O(26) é chamado, uma vez para cada caractere na string da nova ordem lexicográfica.

Para realizar a leitura do texto passado no arquivo de entrada é feita a leitura das linhas até o fim, essa operação tem custo O(n) uma vez para cada linha. Após isso outra operação com custo O(n) é chamada, para passar todos os caracteres da string de texto em seu respectivo minúsculo, em seguida, um laço que remove todos as pontuações e espaços do texto e armazena as palavras em um array é chamado, ele também custa O(n) uma vez para cada caractere na string de texto.

Finalmente, após realizar o QuickSortMain, o programa realiza um laço para calcular o número de ocorrência das palavras, o custo dessa operação aparenta ser O(n^2) por possuir dois laços aninhados, mas na verdade é O(n) pois o valor de incremento do primeiro laço é alterado conforme se itera no segundo laço mais interno.

Leitura dos argumentos O(4) + leitura da ordem lexicográfica O(n) + armazenamento em string com outros caracteres O(26) + tratamento de caracteres maiúsculos O(26) + leitura do texto O(n) + tratamento de caracteres maiúsculos O(n) + remoção de pontuações e armazenamento em array O(n) + cálculo de ocorrência das palavras O(n)

= O(n).

3.1.2 QuickSort

O QuickSort é chamado quando todos os dados já foram lidos e tratados, o quickSort possui alguns laços especiais pela sua condição de parada e chamadas recursivas, isso permite que a complexidade desse método de ordenadas seja O(nlogn), porém, além disso o quickSort conta com o auxílio de outra função para funcionar corretamente. Essa função possui complexidade O(n^2), e será detalhada no próximo tópico, isso torna o QuickSort um pouco mais custoso, sendo ele O((n logn)^2).

QuickSort O((n logn)^2)

3.1.3 ComparaString

Para permitir a ordenação de strings foi implementada uma função que compara duas strings a partir da nova ordem lexicográfica, essa função possui complexidade quadrática, no entanto isso não afeta o funcionamento do programa, pois somente em um caso muito especial de duas strings muito grandes em que se diferenciam apenas nos seus últimos caracteres que ela irá se tornar preocupante.

ComparaString O(n^2)

3.1.4 Inserção

A Fim de reduzir o custo das chamadas recursiva do quickSort que será detalhado melhor mais adiante, um método simples de ordenação também foi implementado, esse método é chamado de inserção e possui dois laços aninhados, em que o laço mais interno também utiliza a função ComparaString, o método inserção possui complexidade O(n^2), mas como conta com a função ComparaString, a complexidade se torna O(n^2^2).

InsertSort O(n^2^2)

Finalmente a função QuisckSortMain possui complexidade

O(((n-s) logn)^2) + O(n^2^2)

No final, tem-se que

O(n) + O(((n-s) logn)^2) + O(s^2^2)

3.2 Espaço

Vamos considerar que cada palavra do texto ocupe uma posição de memória, assim como cada palavra é armazenada apenas uma vez no array de string então o programa ocupa o espaço n palavras, ignorando o espaço que é utilizado na string da ordem lexicográfica que ocupa 37 espaços, mas como é transformada em apenas uma string ocupa 1, e portanto não afeta a análise, além disso com as trocas de string para array durante a execução algumas palavras podem ficar duplicadas nas duas estruturas de dados, mas é liberado rapidamente. Logo, no pior caso, em algum momento da execução existem 2 espaços sendo ocupados por palavras, um na string com todo o texto e outra vez no array de strings. Logo, o programa o ocupa o espaço de O(2n) = O(n).

4. Estrategias de Robustez

Na existência de casos em que as entradas são inválidas foram tomadas medidas contra esse tipo de problema, foi decidido que o programa irá executar independente do valor dos parâmetros de execução estiverem inválidos ou não, nesse sentido, se o parâmetro de execução for invalido o programa irá usar valores padrões para esses parâmetros, assim, impedindo que a saída do programa fique fora do esperado.

5. Conclusão

Após a implementação do programa, pode se notar que havia partes distintas para o desenvolvimento do trabalho. A primeira era utilizar os dados da maneira correta para permitir que fossem processados, a leitura e o tratamento desses dados foi muito importante para o funcionamento devido ao programa nas funções posteriores. Essa segunda parte tinha como objetivo utilizar as entradas para ordenar as palavras com a nova ordem lexicográfica, de maneira eficiente, correta, coesa e funcional. Por fim, utilizar as palavras ordenadas para verificar as ocorrências das palavras se tornou algo simples de se fazer, já que as palavras que apareciam repetidas vezes estavam juntas e só bastava ver quantas vezes elas se repetiam, invés de fazer uma busca por palavras desordenadas, que seriam mais custosas.

O grande esforço, sobretudo para o desenvolvimento da parte de ordenar as palavras, era de fazer isso de um jeito ótimo, para isso utilizar os conhecimentos de algoritmos de ordenação, sobretudo da utilização do QuickSort foi fundamental, e deixar esse método mais robusto com estratégias como escolher um bom pivô e utilizar um método simples de ordenação após um certo tamanho de partição foi essencial para manter o compromisso de fazer isso de maneira ótima.

Além disso, entender como poderiam ser classificadas as palavras a partir de uma nova ordem lexicográfica foi fundamental para que tudo ocorresse como o esperado, a tarefa de classificar palavras observando os caracteres e utilizar uma função que diz exatamente qual palavra fica antes de outra na nova ordem lexicográfica foi extremamente importante

para utilizar os métodos de ordenação com dados diversos como palavras em uma ordem não convencional.

Ao fim desse trabalho, ficou ainda mais evidente a importância dos testes de unidade, da refatoração, da modularização e planejamento do código. Ambas práticas possibilitaram um desenvolvimento mais produtivo/consciente de que, com a implementação de novas funcionalidades ou com a refatoração de trechos do código, as funções permaneciam corretas e funcionais.

Referências

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C + +: Capítulo 3: Estruturas de Dados Básicas . Editora Cengage

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Instruções para compilação e execução

- Acesse o diretorio [TP/];
- Cole o arquivo de entrada, ex: [suaentrada.txt];
- Utilizando um terminal, execute o arquivo [MakeFile] utilizando o seguinte comando: <make>;
- Com esse comando, o programa irá compilar gerando os arquivos .o dentro do diretório [TP/obj/] e o [run.out] dentro do diretório [TP/bin/];
- Com o programa devidamente compilado, utilizando um terminal execute o arquivo [run.out] utilizando o seguinte comando:
  <.bin/run.out -o [NomeSaida.o] -i [SuaEntrada.i] -m [Parâmetro da mediana] -s [Parâmetro da partição]>
- Após a execução desse comando será gerado o arquivo [NomeSaida.o] no diretório raiz do programa.