Trabalho Prático 2 - Analisador Lexigráfico
Universidade Federal de Minas Gerais

Igor Rahzel Colares Galdino
Matricula 2020096255

Julho 2022

1 Introdução

O Trabalho Prático consiste na implementação de um analisador lexicográfico que tem como função contar o número de vezes que uma palavra é repetida em um texto, assim como realizar sua ordenação de acordo com a nova ordem lexicográfica. Essa ordenação deve ser feita utilizando-se o Quicksort, além disso foram implementadas duas otimizações, que são a realização da mediana de **m** elementos para seleção de um pivô e a utilização do método de ordenação por inserção para partições de tamanho menor ou igual a **s**.

A seção 2 desta documentação trata sobre os detalhes da implementação do projeto, mostrando como os arquivos do projeto estão divididos, bem como o papel e a funcionalidade de cada um deles para a solução do problema. Já na seção 3 é feita a análise de complexidade das funções. A conclusão está localizada na seção 4. Na seção 5 encontram-se as referências bibliográficas utilizadas, por fim na seção 6 estão as instruções de compilação e execução.

2 Implementação

2.1 Organização do Código

As pastas do projeto estão organizados da seguinte forma:

TP:
- -src:
  - -Word.cpp
  - -Node.cpp
  - -List.cpp
  - -Sort.cpp
  - -main.cpp
- -include:
  - -Word.h
  - -Node.h
  - -List.h
  - -Sort.h
- -bin:
- -obj:
- -Makefile

2.1.1 Arquivos .cpp

2.1.2 Word.cpp

A pasta **src** contém os arquivos com extensão .cpp. O arquivo *Word.cpp* implementa além dos getters e setters da classe *Word*, o método *WordMatches* que é utilizado para verificar se dois objetos

da classe *Word* são iguais, como o programa é *case insensitive* a string recebida como parâmetro tem todos seus caracteres transformados para letra minuscula e depois é feita a comparação do atributo *word* da classe com a nova string.

2.1.3 List.cpp e Node.cpp

No arquivo *List.cpp* é implementada uma lista encadeada com auxílio da classe *Node*, desenvolvida no arquivo *Node.cpp*, a qual possui somente um construtor que inicializa os apontadores como NULL, para realizar os encadeamentos. Os métodos mais relevantes para essa classe são: *place*, o qual posiciona um apontador para um determinado nó da lista, *InsertLast* que insere um nó no final da lista, e por fim o método *Search* que verifica se uma palavra já se encontra na lista, utilizado-se o método *WordMatches* da classe *Word*, caso ela não seja encontrada ela é inserida ao final da lista(utilizado-se *InsertLast*).

2.1.4 Sort.cpp

Em *Sort.cpp* é desenvolvida a parte de ordenação do programa,vamos então descrever o funcionamento e a ideia central de cada função. O construtor da classe *Sort* realiza a alocação dinâmica do vetor *array* com o tamanho do valor passado como parâmetro.

O método *Insertion(int L,int R)* realiza a ordenação de uma partição de *array* que inicia no índice L e termina no índice R através do método da Inserção, já o método *Insertion(Word *&median, int m)* ordena através da inserção o vetor *median* que será posteriormente utilizado para calcular a mediana de **m** elementos de uma partição.

O método *AllowSwap* funciona de forma análoga ao operador $>$ na comparação de strings, são realizadas comparações carácter a carácter das strings passadas como parâmetro, enquanto os caracteres não forem diferentes ou alguma string chegar ao fim, o valor dos caracteres é baseado na nova ordem lexicográfica informada, caso o carácter de alguma palavra não estiver no intervalo A-Z, os caracteres da palavra são comparados utilizando-se seu valor na tabela ASCII, caso todos os caracteres sejam iguais, isso implica que uma palavra está contida na outra então deve-se avaliar qual das palavras é a de menor(comprimento).

Os métodos *QuickSort*, *sort* e *Partition* são os responsáveis pela ordenação através do *Quicksort*. Inicialmente é feita uma chamada da função QuickSort passando como parâmetro o tamanho do vetor e os valores de **s** e **m**, essa função faz uma chamada de *sort*, que recebe as extremidades do vetor, bem como os valores de **s** e **m**.

O método *sort* inicializa as váriaveis i e j que são utilizadas para percorrer o *array* e compara o tamanho da partição com o parâmetro **s** e verifica se deve usar a ordenação por inserção, caso o tamanho da partição seja maior que **s** é feito uma chamada do método *Partition*, e outras duas chamadas recursivas do método *sort* caso i e j sejam diferentes das extremidades direita e esquerda da partição respectivamente, pois isso implica que haverá mais de um elemento na nova partição que será passada na chamada recursiva.

Por fim o método *Partition* recebe os índices da extremidade esquerda e direita da partição, ponteiros para i e j e o valor de **m**, caso esse valor seja maior que 1 é alocado um vetor de *Word* chamado *median* de tamanho **m** com os **m** primeiros elementos da partição, então esse vetor é ordenado a partir do método *Insertion(Word *&median,int m)* e então o elemento que está na mediana desse vetor é escolhido como pivô, caso **m** menor ou igual a 1 o elemento escolhido como pivô é o elemento que está no meio da partição. Após a escolha do pivô os elementos da esquerda até o meio(máximo) e da direita até o meio(máximo) são comparados ao pivô caso um elemento da esquerda seja maior ou igual ao pivô ele deve ser trocado com um elemento da direita que é menor ou igual ao pivô, esse procedimento é realizado enquanto i e j não se cruzarem. O arquivo main.cpp é discutido na seção 2.3

2.2 Arquivos de cabeçalho

Na pasta include é onde estão localizados os arquivos de cabeçalho com extensão .h, os quais contém as assinaturas dos métodos e atributos das classes a serem implementadas.

O arquivo *Word.h* são definidos os atributos *int occurrences* e *string word*, esses tem como função armazenar o número de repetições da palavra e a própria palavra respectivamente. Já *Node.h* define o construtor de *Node* e os atributos do nó que será utilizado na implementação da lista encadeada, esses atributos são: *Word w* que armazena um objeto da classe *Word* e *Node* next* e *Node* prev* que apontam para o nó anterior e posterior na lista.

O cabeçalho *List.h* define os atributos *int size*, armazena o tamanho da lista, *Node* head*, *Node* tail* utilizados para apontar para o início e fim da lista. Nesse arquivo são também definidos as assinaturas dos métodos da classe *List*.

Sort tem como atributos *Word* array* utilizado para alocar dinamicamente o vetor *array* de acordo com o tamanho do parâmetro *size* passado no seu construtor, *char order[27]* que armazena a nova ordem lexicográfica e *size* que representa o total de palavras armazenadas.

2.3 Funcionamento do Programa Principal

O arquivo main.cpp é o arquivo principal do programa, Nele inicialmente é implementada a função *clearString* a qual elimina os sinais de pontuação do final das palavras. A função *main* realiza a leitura dos argumentos passados pela linha de comando(arquivos de entrada e saída e valor dos parâmetros **s** e **m**. O próximo passo é a leitura do arquivo, onde utiliza-se a função *Search* implementada no arquivo *List.cpp* para inserir novas palavras na lista ou contabilizar suas ocorrências caso já tenham aparecido. Após a leitura do arquivo aloca-se dinamicamente o vetor *array* com tamanho da lista, ou seja, com o tamanho de palavras diferentes que foram lidas e em seguidas cada uma dessas palavras é atribuida a uma posição do *array*, dessa forma o método QuickSort é chamado e as palavras são ordenadas, por fim é feita uma impressão do *array* ordenado.

2.4 Configurações Utilizada

Abaixo são apresentadas as configurações utilizadas para as realizações dos testes do programa:

- Sistema Operacional do Computador: Linux Ubuntu 22.04 LTS
- Linguagem de Programação : C++
- Compilador Utilizado : g++
- Dados do Processador : Intel® Core™ i5-10310U CPU @ 1.70GHz × 8
- Memória RAM : 8 GB Soldado DDR4 2666MHz

2.5 Estratégias de Robustez

As estratégias de robustez adotadas para o programa estão implementadas na função *main* e verificam se o número de letras informadas na nova ordem lexicográfica é igual a 26, caso isso não ocorra a mensagem *"ERRO: NÚMERO DE CARACTERES DE ORDEM INCORRETO"* é impressa no arquivo de saída ,ou se há ao menos um carácter a ser ordenado, se não houver a seguinte mensagem aparece no arquivo de saída *"ERRO: CAMPO TEXTO VAZIO"*.

3 Análise de complexidade

3.1 classe Word

Os métodos, *Word, getWord,setWord,getOcurrences e AddOcurrences* tem complexidade de tempo e espaço igual a $O(1)$, pois realizam apenas operações como atribuição, retorno ou adição.

WordMatches - complexidade de tempo: A complexidade de tempo dessa função é $O(n)$, onde n é o tamanho da string de entrada, pois cada caracter da string é acessado e convertido para letra minúscula.

WordMatches - complexidade de espaço: A complexidade de espaço é $O(1)$, pois é feito um número constante de inicialização de variáveis.

3.2 classe Node

A classe *Node* possui apenas um construtor que inicializa seus ponteiros como NULL, dessa forma a complexidade de tempo e espaço desse método é $O(1)$.

3.3 classe List

Os métodos *List e GetSize* tem ordem de complexidade de tempo e espaço igual a $O(1)$, pois realizam apenas operações constantes.

place - complexidade de tempo: Essa função é $O(n)$, pois no pior caso irá percorrer todos os elementos da lista.

place - complexidade de espaço: Essa função tem complexidade de espaço igual a $O(1)$, pois inicializa apenas um ponteiro para uma variável de tipo *Node* e um contador de tipo inteiro.

GetItem - complexidade de tempo: A complexidade de tempo dessa função é $O(n)$, pois ela faz operações $O(1)$ e realiza uma chamada para o método *place*, então pelas propriedadese na notação assintótica "O grande"temos $O(1) + O(n) = max(O(1), O(n)) = O(n)$.

GetItem - complexidade de espaço: A complexidade de espaço dessa função é $O(1)$, pois inicializa apenas um ponteiro para uma variável de tipo *Node*.

SetItem - complexidade de tempo: A análise feita para esse método é a mesma realizada para a *GetItem*, logo a ordem de complexidade de tempo é $O(n)$.

SetItem - complexidade de espaço: Assim como a complexidade de tempo desse método, sua complexidade de espaço também é análoga à realizada para GetItem resultando em $O(1)$.

InsertLast - complexidade de tempo: A complexidade assintótica desse método é $O(1)$, pois são apenas realizadas uma sequência de operações constantes, ou seja com ordem de complexidade igual a $O(1)$.

InsertLast - complexidade de espaço: A complexidade de espaço dessa função também é $O(1)$, pois é apenas inicializado um novo nó que será ligado ao final da lista.

Search - complexidade de tempo: seja m o tamanho da string recebida como parâmetro e seja n o tamanho da lista. A ordem de complexidade dessa função é $O(mn)$, pois no pior caso, no qual a palavra não está na lista, esta será percorrida até o final e a cada iteração do loop *while* será feita uma chamada para o método *WordMatches*.

Search - complexidade de espaço: A complexidade de espaço dessa função é $O(1)$, pois o número de variáveis inicializadas é constante.

3.4 Sort

Sort - complexidade de tempo: A complexidade de tempo dessa função é ordem de $O(1)$, pois são realizadas um número fixo de atribuições.

Sort - complexidade de espaço: A complexidade de espaço dessa função é $O(n)$, pois é alocado um vetor do tamanho do parâmetro passado para a função.

GetWord - complexidade de tempo: Esse método é $O(1)$, pois apenas realiza uma operção.

GetWord - complexidade de espaço: A ordem da complexidade de espaço é $O(1)$, pois apenas retorna a variável *Word* da posição desejada do vetor.

SetOrder - complexidade de tempo: A complexidade assintótica desse método é $O(n)$, onde n é o tamanho da string de entrada, pois todos os caracteres são convertidos para letra minúscula.

SetOrder - complexidade de espaço: Esse método tem complexidade de espaço $O(1)$, pois apenas transforma as letras maiúsculas da string de entrada em minúsculas e então realiza uma cópia da string modificada para o vetor *order*.

SetWord - complexidade de tempo: A complexidade assintótica desse método é $O(1)$,pois realiza apenas uma atribuição.

SetWord - complexidade de espaço: A complexidade de espaço também é $O(1)$, pois como mencionado na complexidade de tempo esse método realiza apenas uma atribuição.

Insertion(int L, int R) - complexidade de tempo: Seja n o tamanho da partição,a ordem de complexidade desse método é $O(n^2)$, pois o loop externo sempre executa n-1 vezes, já o loop interno, será executado i vezes para cada iteração do loop externo, ou seja, a ordem de complexidade é dada pela progressão aritmética $\frac{n(n-1)}{2} = O(n^2)$.

Insertion(int L, int R) - complexidade de espaço: A complexidade de espaço é $O(1)$, pois é feito um número contante de inicialização de variáveis.

Insertion(Word *&median, int m) - complexidade de tempo: A análise de complexidade desse método é igual à realizada para o método Insertion(int L, int R), pois o funcionamento do algorítimo é o mesmo. Logo ele é ordem de $O(n^2)$.

Insertion(Word *&median, int m) - complexidade de espaço: Pelas razões mencionadas acima a complexidade de espaço desse método é $O(1)$.

Partition - complexidade de tempo: Esse método tem complexidade $O(n)$,pois percorre os n elementos da partição e os organiza em relação ao pivô.

Partition - complexidade de espaço: A complexidade de espaço desse método é $O(m)$,pois a é alocado um valor de tamanho m na memória para se obter a mediana desses elementos.

sort - complexidade de tempo: O pior caso para esse método ocorre quando o pivô escolhido é sempre o menor ou o maior elemento, isso implica que a equação de recorrência seja $T(n)n+T(n-1) = O(n^2)$.

sort - complexidade de espaço: A complexidade de espaço é $O(m)$, pois a função *sort* faz uma chamada para *Partition*.

QuickSort - complexidade de tempo: Esse método realiza uma chamada para o método *sort*, logo sua complexidade de tempo é $O(n^2)$.

QuickSort - complexidade de espaço: Como esse método realiza apenas uma chamada para *sort* sua complexidade de espaço é ordem de $O(m)$.

AllowSwap - complexidade de tempo: A complexidade de tempo desse método é ordem de $O(n)$, onde n é o tamanho da menor string recebida na entrada da função, o pior caso ocorre quando uma string está contida na outra, fazendo com que n caracteres das duas strings sejam comparados.

AllowSwap - complexidade de espaço: A complexidade de espaço é $O(1)$, pois o número de variáveis inicializadas é constante.

3.5 programa principal

clearString - complexidade de tempo: A complexidade de tempo desse método é $O(n)$,onde n é o número de caracteres de sinais de pontuação no final da palavra, pois loop *while* se repete enquanto houver algum sinal de pontuação no fim da palavra.

clearString - complexidade de espaço: A complexidade de espaço é $O(1)$, pois são apenas realizadas um número contante de inicialização de variáveis.

main - complexidade de tempo: A complexidade de tempo da função *main* é $O(n^2)$, pois caso todas as palavras do texto sejam diferentes a sua pesquisa na lista encadeada através da função *Search* será dada pela progressão aritmética $1+2+3+...+n-2+n-1+n = \frac{n(n+1)}{2} = O(n^2)$. Isso ocorre, pois a cada palavra inserida, deve-se percorrer uma palavra a mais em relação a anterior na lista para verificar que ela não foi inserida.

main - complexidade de espaço: A complexidade de espaço é $O(n)$,pois caso todas as palavras do texto sejam diferentes, a lista encadeada possuirá n nós, além disso também será alocado um vetor de n posições para a classe *Sort*.

4 Conclusão

O trabalhou tratou do problema de contar o número de repetições das palavras em um texto, além da ordenação destas, por meio do Quicksort,em uma nova ordem lexicográfica fornecida. A abordagem incial para a solução desse problema foi a criação da classe *Word* a qual possuía atributos e métodos voltados para realizar a contagem das repetições da palavra, bem como realizar o armazenamento da palavra nos padrões especificados. Como o número de palavras no texto não é especificado, uma forma encontrada para contornar esse problema foi a criação da classe *List* a qual implementa uma lista encadeada com o objetivo de armazenar as palavras diferentes presentes no texto e realizar a contagem das palavras repetidas. Por fim foi implementada a classe *Sort*, que tinha como papel tratar da ordenação das palavras lidas, nessa classe foram implementados os métodos de ordenação como Quicksort e Inserção, além de uma função (*AllowSwap*) que serviu como o operador ¡ para verificar se uma palavra era maior ou menor que a outra baseado na nova ordem lexicográfica.

O principal desafio do trabalho foi a implementação do Quicksort juntamente com a otimizações propostas, pois foi necessário compreender de forma mais aprofundada o algoritmo e elaborar maneiras de como essas otimizações seriam inseridas nele. Outra parte do trabalho prático que teve de ser analisada com cuidado foi a implementaçãp o da função *AllowSwap*, pois haviam vários detalhes que deveriam ser levados em conta, por exemplo o caso de algum dos caracters não pertencerem ao intervalo A-Z, o que implicava em compara-los utilizando a tabela ASCII.

5 Referências

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

https://www.cplusplus.com/reference/fstream/fstream/

6 Instruções de Compilação e Execução

- Pelo terminal acesse o diretório da pasta TP
- Execute o arquivo *Makefile* utilizando o comando *make all*
- Esse comando irá gerar o arquivos objetos .o dos arquivos .cpp
- Proceda para digitar o comando *.\bin\tp1.out*