Documentação TP2

Aluno: Henrique Daniel de Sousa

Matrícula: 2021031912

Introdução

Essa documentação tem por objetivo apresentar um programa que desenvolve um analisador de texto, no qual são contadas quantas vezes cada palavra do texto aparece, sendo que depois elas são ordenadas mediante a uma ordem lexicográfica predeterminada pelo usuário do programa.

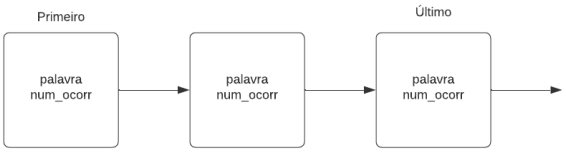
Implementação

O código está organizado entre arquivos ".h", que implementam os TADs "palavra" e "texto", e arquivos ".cpp" que implementam as funções de cada TAD e as executam. A execução das funcionalidades criadas é feita no arquivo "analisador.cpp".

TAD Palavra: Uma palavra é definida por uma string *"palavra"* e um inteiro *"num_ocorr"*, que guarda a quantidade de vezes que essa palavra foi encontrada no texto lido. Além disso, há, também, um ponteiro do tipo *"Palavra"* que aponta para a próxima palavra. Ou seja, o tipo *"Palavra"* é uma célula de uma lista encadeada.

TAD Texto: O texto é uma lista encadeada que possui o tipo *"Palavra"* como célula. Ele é definido por um ponteiro que determina a primeira posição da lista, outro que determina a última posição e o número de palavras distintas inseridas na lista. Além disso, há um valor inteiro que representa o id da lista, para fins de gerar gráficos de registro de memória.

Dessa forma, a estrutura de dados implementada segue o seguinte modelo:



O programa recebe como parâmetros:

1. O nome de um arquivo de entrada, constituído por uma ordem lexicográfica, sendo esta composta por 26 caracteres, e um texto. Dessa forma, o texto é lido e, para cada palavra obtida verifica-se se ela já foi previamente inserida na lista ou não. Caso tenha sido, o número de ocorrências dela é incrementado em um, caso contrário, a nova palavra é inserida na lista.

2. Dois parâmetros 'm' e 's'.

    a. 'm' determina a estratégia de pivoting para o algoritmo de ***quick sort***, que no caso será a mediana de m elementos. Caso o tamanho da partição seja menor que o valor de m, o pivô é definido pela mediana de 1. Se m <= s, então a mediana de m elementos é feita ordenando os m elementos da posição do meio da partição usando o ***insertion sort***. Caso m > s, usa-se a mediana de 1.

    b. 's' determina o tamanho máximo da partição para que seja usado um algoritmo de ***insertion sort*** para a ordenação desta partição. Logo, quando o comprimento da partição for menor ou igual a 's', o algoritmo de ordenação será ***insertion sort.***

3. Um arquivo de saída, no qual são impressas cada palavra e seu número de ocorrências.

4. Um arquivo para salvar o registro de acesso e o tempo de execução do programa.

A implementação desse problema foi realizada usando a linguagem C++ e compilada com o G++. Para a documentação desse problema, a máquina utilizada tem o Windows10(WSL2) como sistema operacional, um processador Intel(R) Core(TM) i5-9300H CPU @2.40GHz 2.40 GHz e 8GB de memória RAM.

Análise de Complexidade

Inserção de palavras:

Função *'Texto::insere()'*: Essa função insere uma palavra na lista encadeada. Portanto, sua complexidade de tempo é O(n) e de espaço é O(1).

Ordenação das palavras:

Função *'Texto::insertionSort()'*: Essa função implementa o algoritmo de insertion sort, o qual tem como complexidade de tempo O(n) quando a entrada já estiver ordenada e $O(n^2)$ caso contrário. Sua complexidade de espaço é O(1).

Função *'Texto::find_pivot()'*: essa função recebe os parâmetros m, s, low e high. 'm' e 's' são passados na entrada do programa e low e high determinam a primeira e a última posição da partição. Caso a partição tenha comprimento menor do que m, então a complexidade de tempo da função será O(1), pois o pivô será a mediana de 1. Caso m seja maior que s, a complexidade de tempo também será O(1). Porém, se m for menor ou igual a s, a complexidade de tempo será $O(m^2)$, pois será utilizada a função de insertion sort para ordenar o subvetor com m elementos e obter a mediana desse subvetor ordenado. Desse modo, obedecendo a condição de m <=s , obtêm-se a mediana de m elementos como o pivô do quick sort.

Função *'Texto::quicksort()'*: Essa função implementa o algoritmo de quicksort híbrido, ou seja, o quicksort é utilizado até que a partição atinja um valor menor ou igual ao parâmetro 's', passado na entrada do programa. Desse modo, o quicksort possui complexidade de tempo O(n log n) e o insertionSort possui complexidade de tempo $O(n^2)$. Desse modo, caso s = n, a complexidade da função será $O(n^2)$ e a complexidade de espaço será de O(log n).

Busca e acesso:

Função *'Texto::posiciona()'*: Essa função caminha pela lista até na posição informada nos parâmetros. Logo, ela tem O(n) como complexidade de tempo e O(1) como complexidade de espaço.

Função *'Texto::acessaPalavras()'*: Essa função acessa todas células da lista para fins de registro de acesso à memória. Logo ela tem O(n) como complexidade de tempo e O(1) de complexidade de espaço.

Impressão dos resultados:

Função *'Texto::acessaPalavras()'*: Essa função acessa todas células da lista para a impressão dos resultados. Logo ela tem O(n) como complexidade de tempo e O(1) de complexidade de espaço.

Estratégias de robustez

O programa usa a biblioteca "msgassert.h" para implementar a robustez e corretude das entradas recebidas.

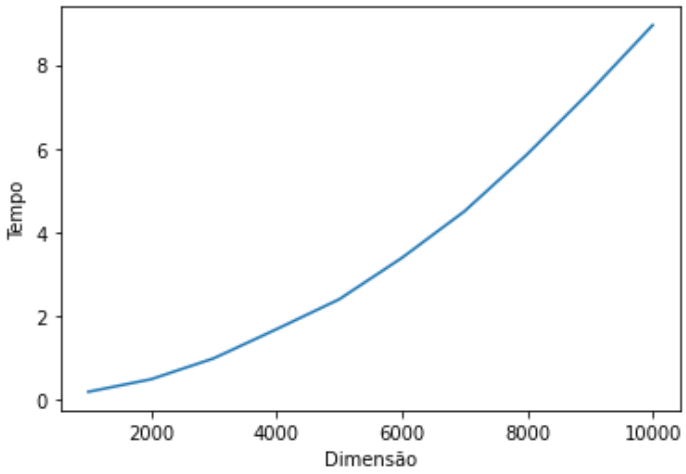
Análise experimental

Desempenho computacional

Para a análise de desempenho computacional foram utilizadas entradas com 1000 palavras até 10000 palavras, contando de 1000 em 1000. O valor de m utilizado foi 3 e de s foi 10. As palavras foram geradas de modo aleatório.

Observa-se então, de acordo com o tempo de execução e com as dimensões, o seguinte gráfico:

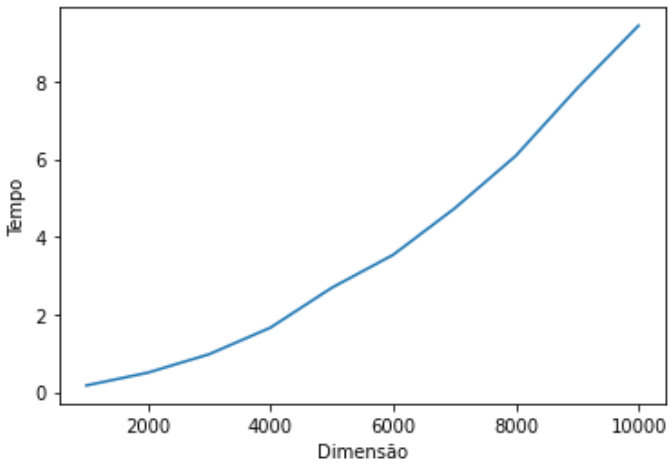
Dimensão	Tempo
1000	0.1835186
2000	0.4834176
3000	0.9830436
4000	1.6786157
5000	2.3932972
6000	3.3831772
7000	4.4984502
8000	5.8615994
9000	7.3570851
10000	8.9456079



"Gráfico gerado usando o tempo de execução do programa e o número de palavras, por meio das funções da biblioteca 'matplotlib', em Python"

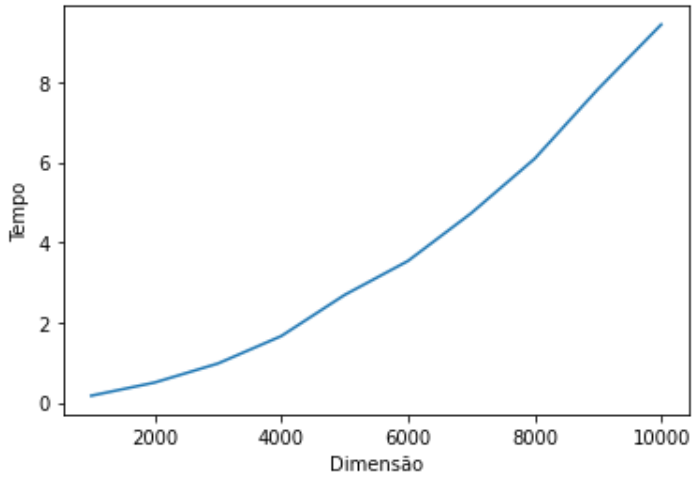
Mudando o valor de s para 5 e utilizando as mesmas entradas, obtém-se os seguintes resultados:

Dimensão	Tempo
1000	0.1741587
2000	0.5022689
3000	0.9773633
4000	1.6622169
5000	2.6893126
6000	3.53675
7000	4.7306133
8000	6.0882601
9000	7.8242923
10000	9.4401106



Mudando o valor de s para 3 e utilizando as mesmas entradas, obtém-se os seguintes resultados:

Dimensão	Tempo
1000	0.1864149
2000	0.5051335
3000	0.998572
4000	1.6442652
5000	2.5677664
6000	3.4199205
7000	4.7202471
8000	6.1278048
9000	7.7746796
10000	9.4940287



Desse modo, é possível perceber que, para uma quantidade maior de palavras, o parâmetro s = 10 comportou-se de maneira mais eficiente. Contudo, para quantidades menores de palavras, s = 5 teve melhores resultados.

Observa-se, então, que o programa tende a uma complexidade de O(n log n), mesmo utilizando um algoritmo de quick sort "híbrido" com insert sort, o que otimiza o desempenho do programa. A função que utiliza maior parte do tempo é a função *"posiciona()"* que tem complexidade de tempo O(n) e é usada para caminhar pela lista até a posição informada como parâmetro. Isso pode ser observado na seguinte análise do gprof, retornada a partir de uma entrada com 10000 palavras e parâmetros m como 3 e s como 10:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 94.69      1.57     1.57   148557     0.00     0.00  Texto::posiciona(int, bool)
  3.02      1.62     0.05 12588037     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char
  1.21      1.64     0.02      934     0.00     0.00  Texto::partition(int, int, int*, int*
  0.60      1.65     0.01    90537     0.00     0.00  comparePalavra(Palavra*, Palavra*, ch
  0.60      1.66     0.01        1     0.01     0.06  Texto::analisaTexto(std::basic_ifstre
  0.00      1.66     0.00   300446     0.00     0.00  std::char_traits<char>::compare(char
  0.00      1.66     0.00    90537     0.00     0.00  unsigned long const& std::min<unsigne
  0.00      1.66     0.00    37487     0.00     0.00  bool std::operator==<char, std::char_
  0.00      1.66     0.00    36235     0.00     0.00  Palavra::~Palavra()
  0.00      1.66     0.00    31235     0.00     0.00  Palavra::Palavra()
  0.00      1.66     0.00    28543     0.00     0.00  swap(Palavra*, Palavra*)
  0.00      1.66     0.00     7666     0.00     0.00  Palavra::operator=(Palavra const&)
  0.00      1.66     0.00     5000     0.00     0.00  lower(std::__cxx11::basic_string<char
  0.00      1.66     0.00     5000     0.00     0.00  Texto::insere(std::__cxx11::basic_str
  0.00      1.66     0.00     5000     0.00     0.00  Palavra::Palavra(std::__cxx11::basic_
  0.00      1.66     0.00     2485     0.00     0.00  std::operator|(std::_Ios_Openmode, st
  0.00      1.66     0.00     1757     0.00     0.00  Texto::insertionSort(char*, int, int)
  0.00      1.66     0.00      934     0.00     0.00  Texto::find_pivot(int, int, int, int,
  0.00      1.66     0.00        2     0.00     0.00  defineFaseMemLog(int)
  0.00      1.66     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z5lowerNSt7__cxx1112b
  0.00      1.66     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN7PalavraC2ENSt7__cx
  0.00      1.66     0.00        1     0.00     0.00  _GLOBAL__sub_I_regmem
  0.00      1.66     0.00        1     0.00     0.00  parse_args(int, char**, char*, char*,
  0.00      1.66     0.00        1     0.00     0.00  clkDifMemLog(timespec, timespec, time
  0.00      1.66     0.00        1     0.00     0.00  iniciaMemLog(char*)
  0.00      1.66     0.00        1     0.00     0.00  desativaMemLog()
  0.00      1.66     0.00        1     0.00     0.00  finalizaMemLog()
  0.00      1.66     0.00        1     0.00     0.00  ordemLexicografica(char*, std::basic_
  0.00      1.66     0.00        1     0.00     0.00  __static_initialization_and_destructi
  0.00      1.66     0.00        1     0.00     0.00  __static_initialization_and_destructi
  0.00      1.66     0.00        1     0.00     0.00  __static_initialization_and_destructi
  0.00      1.66     0.00        1     0.00     0.00  Texto::acessaPalavras()
  0.00      1.66     0.00        1     0.00     0.00  Texto::imprimePalavras(std::basic_ofs
  0.00      1.66     0.00        1     0.00     0.00  Texto::limpa()
  0.00      1.66     0.00        1     0.00     1.60  Texto::quicksort(int, int, char*, int
  0.00      1.66     0.00        1     0.00     0.00  Texto::Texto()
  0.00      1.66     0.00        1     0.00     0.00  Texto::~Texto()
```

"Saída gerada pelo "gprof", aplicando uma entrada com 10000 palavras."

Registro de memória e localidade de referência

Para a análise do registro de memória foi utilizada a biblioteca *'memlog.h'* e programa *'Analisamem'*.

Assim o gráfico de acesso de memória gerado, a partir da seguinte entrada:

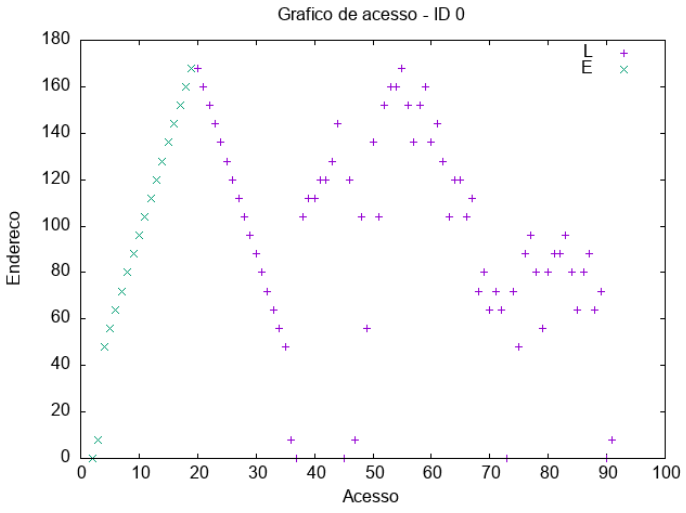
#ORDEM

Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

#TEXTO

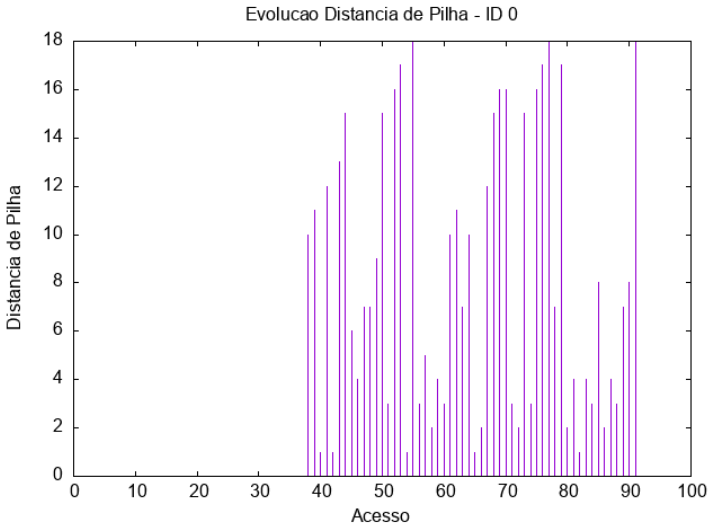
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis felis ex, facilisis ac leo et, suscipit luctus purus.

O valor de m e de s foi, de ambos, 3.



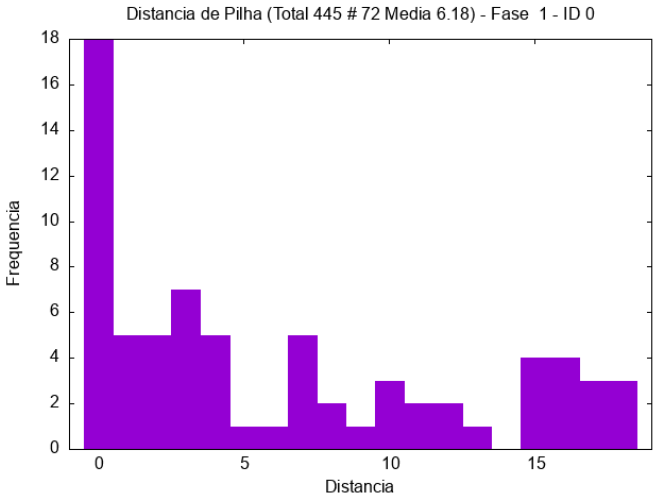
"Gráfico gerado usando o programa gnuplot

Esse gráfico representa a alocação de memória da lista encadeada implementada e seus acessos. Desse modo, é possível ver que a inserção na lista é feita sempre no início, pois a última célula inserida foi a primeira acessada. Depois observa-se o algoritmo de quick sort híbrido, onde na primeira parte tem-se o quick sort e posteriormente, quando as partições tiverem seu comprimento menor ou igual a s, o insertion sort é utilizado.



"Gráfico gerado usando o programa "gnuplot"

Observa-se que a distância entre os endereços de cada posição acessada varia muito, ou seja, a distância de pilha tem valores muito variados. Isso pode ser visto no seguinte gráfico:



"Gráfico gerado usando o programa "gnuplot", a partir da análise de distância de pilha total."

Conclusão

Os principais temas abordados foram a abstração da palavra como um TAD Palavra, e o texto como um TAD Texto e como ordenar as palavras usando um algoritmo de quick sort híbrido. Na primeira parte, foi implementada uma lista que armazena as diferentes palavras presentes no texto de forma coesa e contando o número de repetições da palavra. A segunda parte, por sua vez, consta na implementação de um método de ordenação para ordenar as palavras de acordo com uma ordem lexicográfica pré-estabelecida. Para isso, foi necessário, além do algoritmo de ordenação propriamente dito, que no caso foi o quick sort, um método para comparar as palavras, de acordo com a ordem lexicográfica passada.

Observa-se que há várias formas de otimizar o quick sort, por exemplo, utilizando a mediana de m números. Isso evita o pior caso de ocorrer, o qual teria complexidade de tempo $O(n^2)$, ou seja, muito maior do que O(n log n), que foi a aproximação obtida a partir das experimentações. Além disso, para uma partição menor que o parâmetro s informado, um algoritmo de insertion sort se mostra mais eficiente. Logo, o quick sort híbrido usa o quick sort e o insertion sort, o que otimiza a execução do programa.

Ao fim deste trabalho ficou evidente a importância da implementação de uma estrutura de dados coerente e que possa abstrair ao máximo as informações passadas pelo texto. Além disso, dentre os métodos de ordenação , o quick sort possui grandes vantagens em relação ao tempo de execução e deve ser utilizado preteritamente, sempre que possível.

Bibliografia

https://towardsdatascience.com/an-overview-of-quicksort-algorithm-b9144e314a72

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT.

Instruções para compilação e execução

1. Acesse o diretório chamado **‘TP’**.
2. Adicione o arquivo de entrada nesse diretório.
3. No Makefile, modifique o nome da entrada, em **‘ARQIN’** para o nome de seu arquivo de entrada, o nome da saída em **‘ARQOUT’**, o valor de -m e de -s em **‘M’** e **‘S’**.
4. No terminal, execute o Makefile com o comando make. A saída será gerada no diretório **‘TP’**.