Trabalho Pratico 2

Analisador

João Vitor Ferreira

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

joaovitorferreira@ufmg.br

1. Introdução

O problema proposto foi de implementar um código no qual realiza a leitura de um texto de um arquivo de entrada e o ordena seguindo uma norma alfabética que é passado nesse arquivo e por gera a saída ordenada segunda essa nova ordem juntamente da contagem de repetições de cada palavra no texto.

2. Método

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

2.1. Estrutura de Dados

A implementação teve como base o uso de 1 classe que armazena os dados relevantes a palavra.

Esta classe possui o nome de Analisador, e nela são guardadas as seguintes variáveis:

string palavra_do_texto – Nesta variável é armazenada a palavra lida do arquivo

int lexicografica_da_ordem[1000] – Neste array de in são armazenados os pesões no novo alfabeto de cada caractere da palavra. Ele está assumindo que nenhuma palavra sera maior do que 1000 caracteres.

int id – Um id virtual criado para cada palavra do texto

int repeticoes – A variável que guarda a quantidade de repetições de cada palavra no texto.

int simples = 0 – Guarda o valor da partição em que o é necessário usar o algoritmo simples de seleção.

int contado = 0 – Guarda e seta se a palavra já foi contada.

Além das variáveis foram implementados dois bool operator o de > e <. Ambos possuem a mesma lógica porém os retornos são diferentes para cada um deles acordando com o tipo. Dessa forma somente é necessário detalhar 1 e o funcionamento do outro é análogo, dessa forma irei detalhar o bool>

bool operator>(Analisador Analisador) – É usado na comparação de dos Analisadores, primeiramente ele seta duas variáveis com o tamanho de cada palavra. Depois ele realiza um loop no qual deve rodar até que as duas palavras acabem, e há testes de comparação em caso da letra de posicao i ser igual nas duas palavras a próxima letra sera analisada, em caso de letras diferentes, se a letra “a” vier depois de “b” é retornado falso, se a letra “a” vier antes de “b” é retornado verdadeiro, em caso da palavra “a” acabar e “b” não é retornado falso e no caso de “b” acabar e “a” não é retornado falso.

Foram criadas 10 funções sendo 10 delas no Analisador.hpp e 1 no Execucao.hpp

Executa.hpp

- void Executa(int Mediana, char *arquivo_entrada, char *arquivo_saida, int Simples) – Recebe o valor da Mediana, o nome do arquivo de entrada e saída, e o valor da partição para usar o algoritmo Simples. Nesse arquivo o programa é inicializado e a leitura do arquivo de entrada se inicia, o vetor de palavras é criado, e o texto é adicionado a cada posicao deste array, ao longo da adição as conversões são realizadas, como converter todas as letras da palavra para minúsculo, remover caracteres especiais. Após a leitura do texto ser realizada é iniciada a leitura da nova ordem lexicográfica, as letras são adicionadas num

array de char de tamanho igual ao do alfabeto tradicional. Após o arquivo de entrada terminar se inicia os tratamentos de ordenação das palavras é chamada a função de atribui os pesos das letras das palavras, a busca de repetição das palavras, o método de ordenação que é o QuickSort, a escrita da ordenação no arquivo e por fim o arquivo de entrada é fechado e o array de palavras é desalocado da memória que foi previamente dedica a ele. Terminando assim a função.

Analisador.cpp

- void AtribuiOrdem(Analisador Analisador[], int qtde_palavras, char Letras[]) – O Analisador, a quantidade de palavras e o array de letras são recebidos. Nessa função ele percorre cada palavra, em cada palavra cada letra e compara cada letra de cada palavra com as letras do no alfabeto afim de encontrar a correspondência e adicioná-la no array lexicografica_da_ordem na posicao daquela letra o peso que ela representa no novo alfabeto
- void ConverteMinusculo(Analisador Analisador[], int pos_palavra, char palavra_extraida[]), recebe o analisador a posicao da palavra e a palavra extraída, analisa se no final da palavra há um caractere que não seja uma letra ou número, caso ele seja ele é removido e pôr fim a nova palavra é passada para a palavra no array.
- void BuscaRepeticao(Analisador Analisador[], int qtde_palavras) – Recebe o Analisador, a quantidade de palavras, realiza 2 loops encadeados e compara cada palavra com todas as outras com o intuito de procurar repetições. Caso seja encontrada uma variável auxiliar é acrescida e o fator contado é alterado para que essa palavra não seja contada novamente.
- void Saida(Analisador Analisador[], int qtde_palavras, char *nome_saida) - Recebe o Analisador, a quantidade de palavras e o nome do arquivo de saída. Realiza a abertura do arquivo e o preenche com os resultados, que são as palavras ordenadas segundo a ordem passado inicialmente e com a sua quantidade de repetições.
- void Selecao(Analisador Analisador[], int final, int inicio) – Recebe o Analisador, a posição final e inicial das palavras. Ele executa o Algoritmo de seleção, o código usado é o mesmo apresentado em aula.

- void Particao(char Esq, char Dir, int *i, int *j, Analisador *Analisador) Recebe o Analisador, a posição final e inicial das palavras. Ele executa o Algoritmo de QuickSort Recursivo, o código usado é o mesmo apresentado em aula.
- void QuickSort(Analisador *Analisador, int n) – Recebe o Analisador, a posição final das palavras. Ele executa o Algoritmo de QuickSort Recursivo, o código usado é o mesmo apresentado em aula.
- void Ordena(int Esq, int Dir, Analisador *Analisador) - Recebe o Analisador, a posição da direita e esquerda das palavras. Ele executa o Algoritmo de QuickSort Recursivo, o código usado é o mesmo apresentado em aula.

Partida contém somente os comandos que são necessários para que a analise seja executada, como a leitura do arquivo, a chamada das funções, é feito numa função denominada void Executa().

Main, há o main do programa nele é iniciado e finalizado a Analise de Complexidade e chamado a função Executa, além de analisar os comandos passos pela linha de comando e convertidos para variáveis que serão usadas no programa.

2.2. Classes

A modularização foi implementada ao longo do código, por tanto foi necessário a criação de 1 classe, esta que foi usada para somente armazenar os atributos de cada palavras assim como abrigar o bool operator necessário para o funcionamento do algoritmo de ordenação usado o QuickSort.

2.3. Formato de Entrada e Saída

O formato de entrada é a inserção de uma pasta denominada entrada a qual conterá os arquivos com as entradas para o programa, o nome deste arquivo deve ser passado. A saída é padronizada e estará contida no arquivo de nome a ser determinado e está na pasta denominada saída.

3. Analise de Complexidade

3.1. Tempo

Inicialmente iremos seguir algumas preposições as estruturas auxiliares mais básicas consideraremos com o $O(1)$, e o custo da funções são:

void Executa(int Mediana, char *arquivo_entrada, char *arquivo_saida, int Simples) – A função possui diversas atribuições e operações das quais não iremos considerar seu tempo de execução. Somente iremos considerar o while como função que afetara o tempo portanto como o while depende estritamente do tamanho da entrada temos que essa função tem custo $O(n^2)$

void AtribuiOrdem(Analisador Analisador[], int qtde_palavras, char Letras[]) – A função possui 3 loops aninhados, portanto, intuitivamente o custo dele parece ser $O(n^3)$ porém o ultimo loop possui o tamanho fixo, igual a 25, então ele custa $O(1)$, então temos que o custo total dessa função é $O(n^2)$

void ConverteMinusculo(Analisador Analisador[], int pos_palavra, char palavra_extraida[]) – são feitas comparações nesta função mas elas não têm um impacto significativo na análise que estou performando. Dessa forma o for compor sozinho o custo dela que é de $O(n)$

void BuscaRepeticao(Analisador Analisador[], int qtde_palavras) – Nesta função temos 2 loop aninhados que dependem de n, logo temos custo igual a $O(n^2)$

void Saida(Analisador Analisador[], int qtde_palavras, char *nome_saida) – Nesta função o custo de escrita no arquivo sera desconsiderado restando somente o custo do loop para realizar essa escrita que sera de $O(n)$

void Selecao(Analisador Analisador[], int final, int inicio) – O custo da função de seleção é $O(n)$

void Particao(char Esq, char Dir, int *i, int *j, Analisador *Analisador), void QuickSort(Analisador *Analisador, int n), void Ordena(int Esq, int Dir, Analisador

*Analisador) – O custo da função QuickSort é variável, nesta aplicação calculo que em geral seu custo seguira a quantia de $O(n \log n)$, porem como o texto pode estar ordenado inicialmente de diversas maneiras e a ordem alfabética é dinâmica não podemos descartar o pior caso que é $O(n^2)$. Dessa forma esperamos que o caso media seja mais frequente que o pior caso do QuickSort, porém como não temos controle nenhum sobre a entrada consideraremos o pior caso $O(n^2)$ como o mais frequente.

3.2. Espaço

O espaço utilizado é variável e dependera da quantidade de palavras que estão no texto, considerando um valor x de palavras sera alocado um quantidade x de posições, no entanto cada palavra possui y letras que serão desconsideradas, dado que o vetor de letras já está previamente definido, x posições de memória especificas somente para guardar os dados dessa palavra. Considerando que em algumas funções são criados locais para armazenas estes conteúdos, no entanto ao realizar uma analisa mais detalhada notamos que, na grande maioria das funções e manipulações são feitas nos locais já previamente estabelecidos, usando assim de um espaço menor. Desta forma desconsiderarei o espaço usado por manipulações que utilizem de espaço que não foram previamente alocados e que minhas funções somente utilizem do espaço alocado inicialmente, então desta forma temos que o espaço é de $\theta(n)$.

4. Estratégias de Robustez

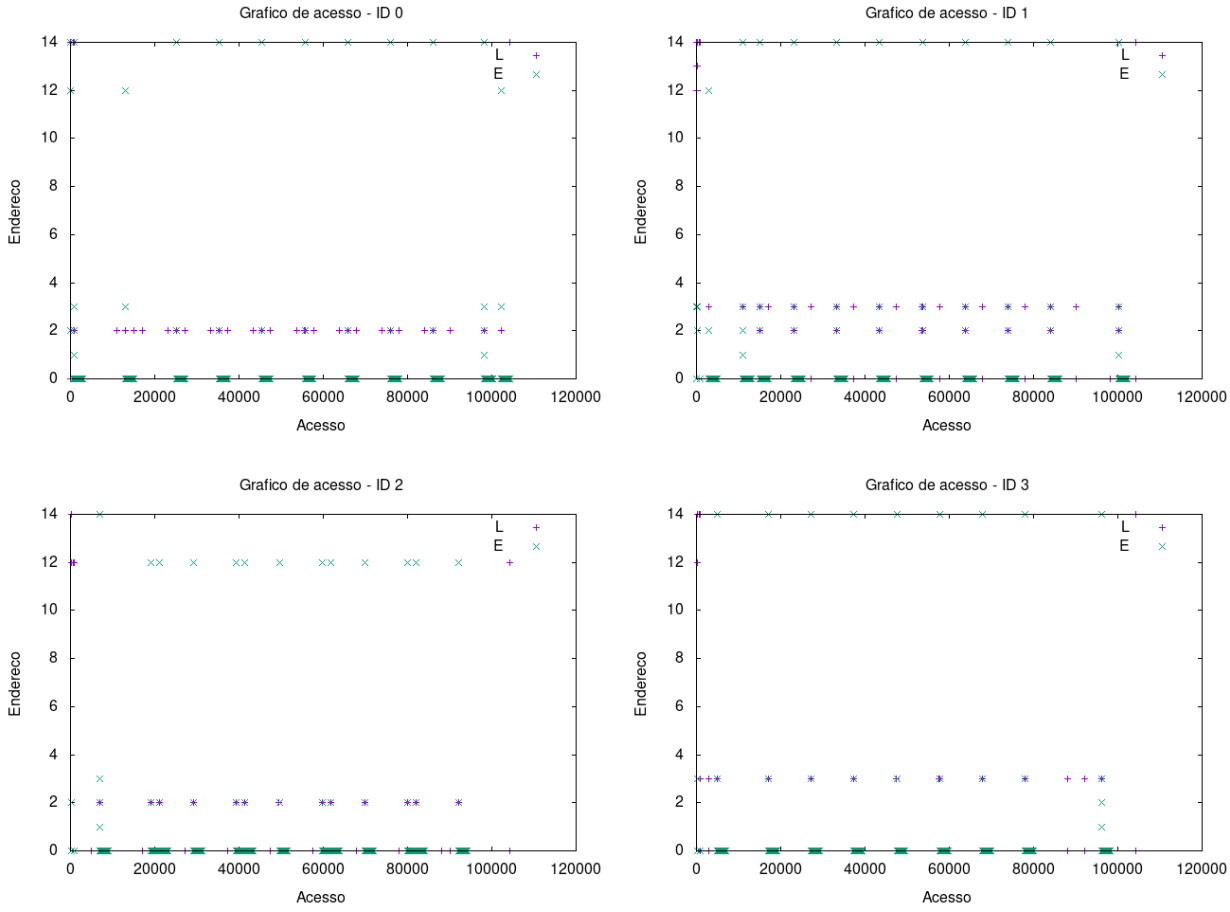
Para garantir o correto funcionamento do código foram adicionados errosAssert, dos quais param o código toda vez que um erro é localizado. Os erros que foram adicionados estão relacionados com a abertura dos arquivos e parâmetros passados invalidamente. No caso da abertura do arquivo caso ele está vazio um erro retornara avisando que não há nada no arquivo ou que ele não pode ser aberto. No caso dos parâmetros temos erros para todos que não são passados corretamente seguindo as instruções, ou seja, o programa sera abortado em caso de valores menores que 0 para

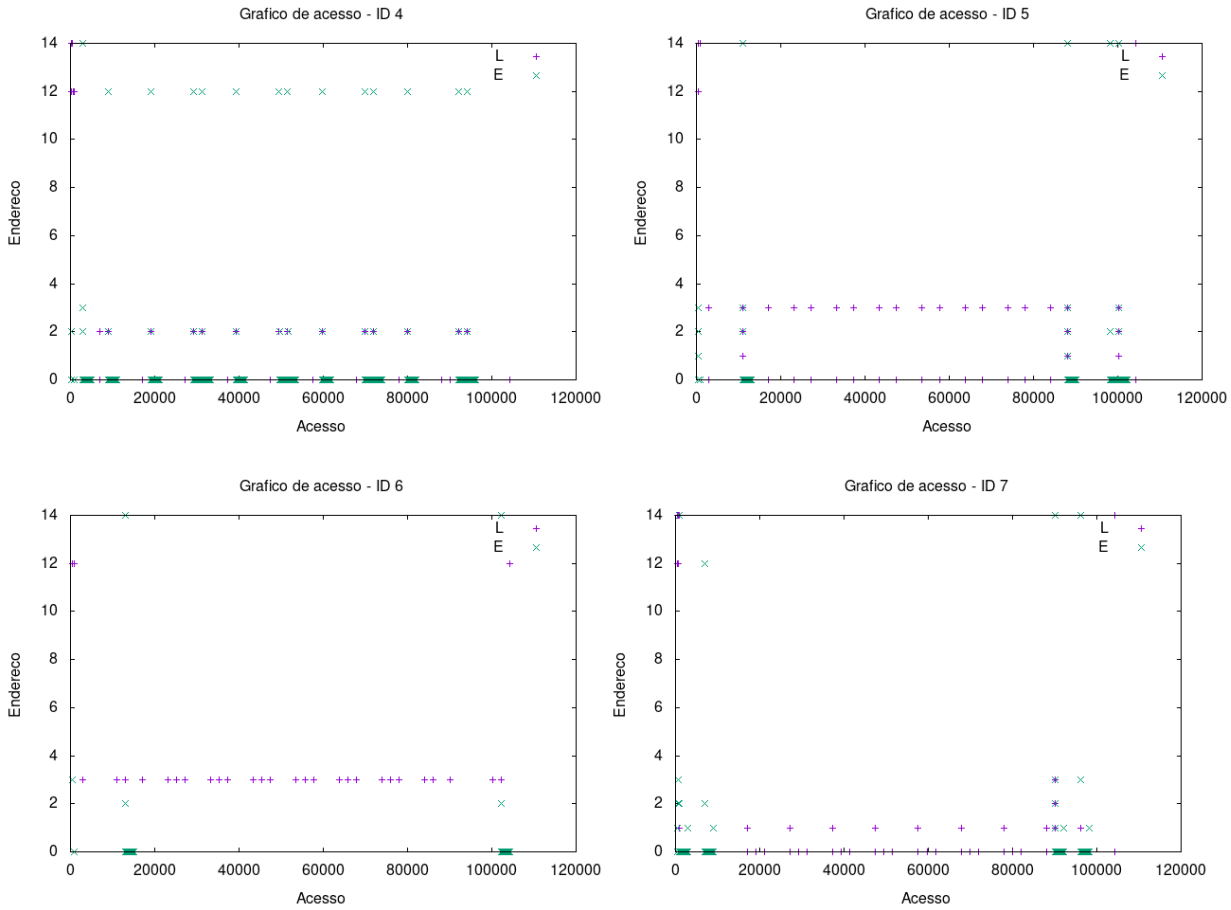
a partição ou seleção, sera dado um aviso caso o tamanho do elemento de seleção seja maior que 30, visto que para valores maiores que esse o algoritmo de QuickSort é mais eficiente.

5. Análise Experimental

Para realizar a análise utilizamos o gerador de carga disponibilizado no Moodle da disciplina, o “geracarga”. Foi gerado um texto contendo 8 palavras. Os mapas de acesso à memória desse caso, além dos tempos tomados pelo programa e a distância de pilha. Para cada um desses testes foi gerado uma derivação para cada palavra do texto.

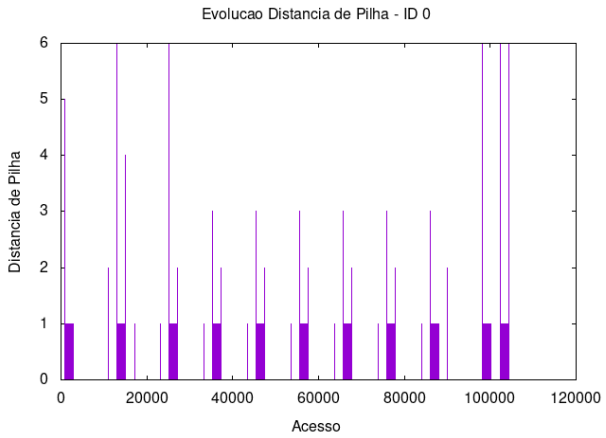
6.1 Mapas de Acesso

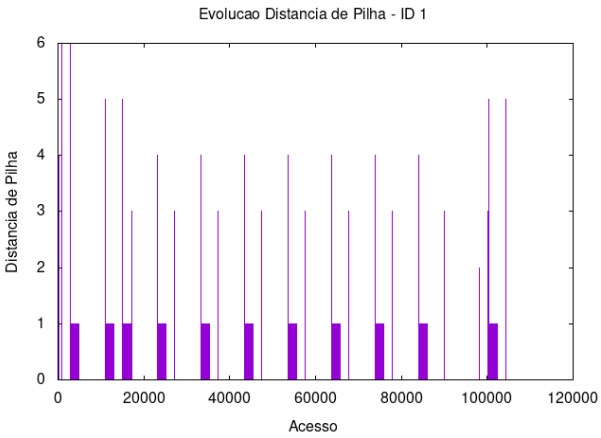


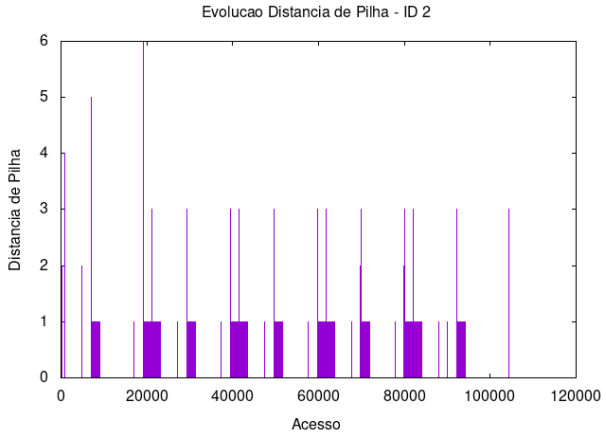


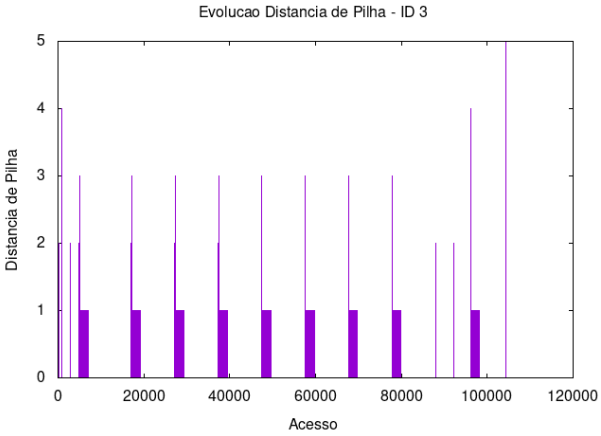
Analisando os gráficos de uma maneira mais geral é notório a forma em que a memória é acessada para leitura e escrita deriva exatamente da forma em que o programa foi desenvolvido. Nota-se que os acesso são em geral consecutivos, especialmente a escrita que em praticamente todos os casos é sempre feita em endereços contínuos, já a leitura tende a ser um pouco mais esparsa, é possível notar certa continuidade nos gráficos se analisarmos áreas especificas de todos juntamente, vemos que o programa faz a alocação continua. O que resulta num acesso a memória “padronizado”. Outra questão notável é continuidade de escrita já que todos as palavras estão num array e sua escrita de dados é dada sequencialmente e unindo todos os gráficos é possível analisar claramente essa continuidade de acesso.

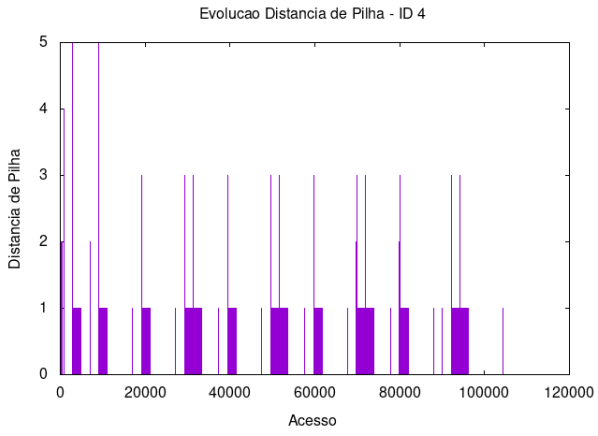
6.1.2 Evolução Distância de pilha

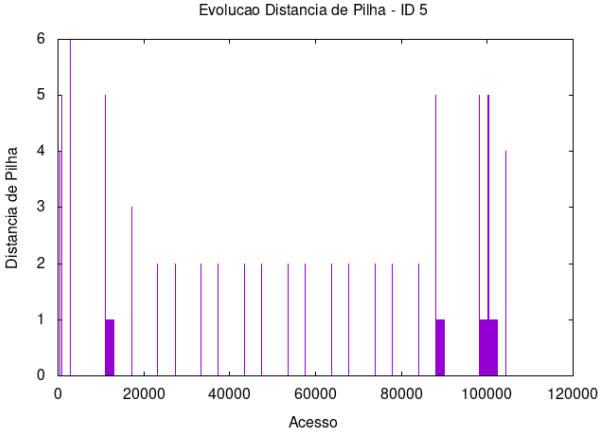

Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
0
20000
40000
60000
80000
100000
120000

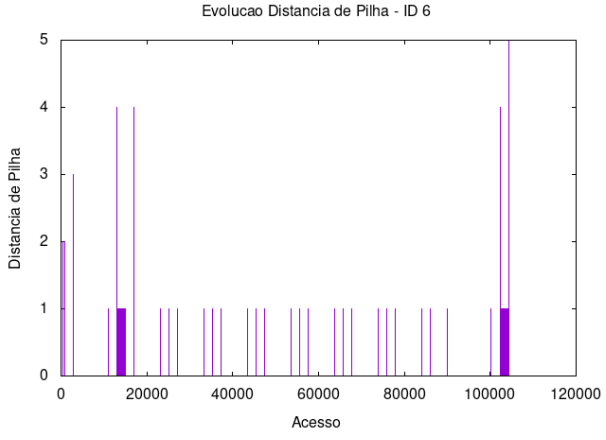

Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
0
20000
40000
60000
80000
100000
120000

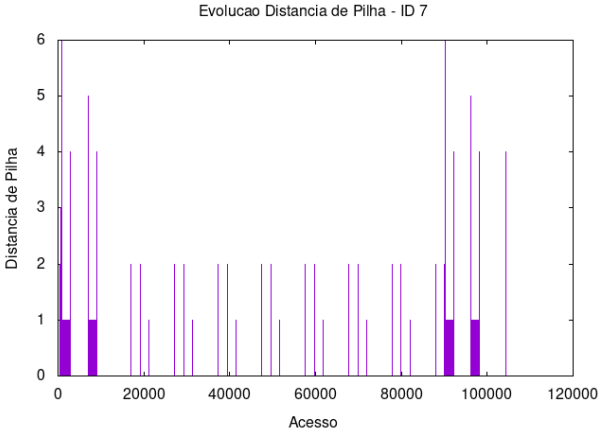

Evolucao Distancia de Pilha - ID 2
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
0
20000
40000
60000
80000
100000
120000


Evolucao Distancia de Pilha - ID 3
Distancia de Pilha
Acesso
0
1
2
3
4
5
0
20000
40000
60000
80000
100000
120000


Evolucao Distancia de Pilha - ID 4
Distancia de Pilha
Acesso
0
1
2
3
4
5
0
20000
40000
60000
80000
100000
120000

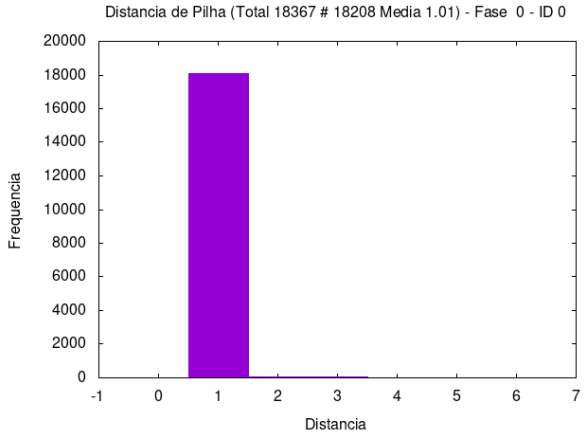

Evolucao Distancia de Pilha - ID 5
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
0
20000
40000
60000
80000
100000
120000

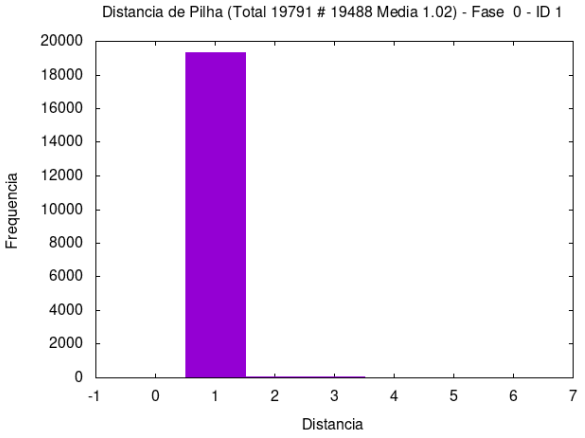

Evolucao Distancia de Pilha - ID 6
Distancia de Pilha
Acesso
0
1
2
3
4
5
0
20000
40000
60000
80000
100000
120000

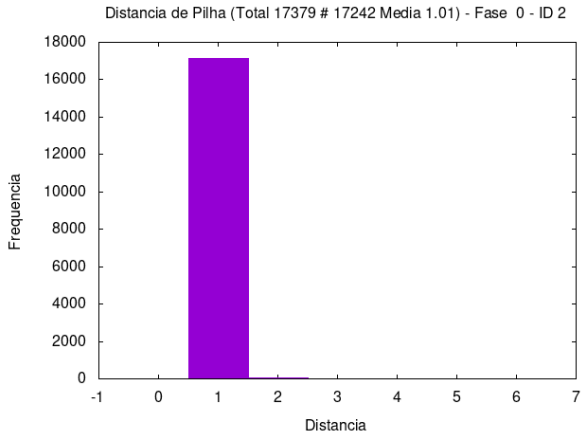

Evolucao Distancia de Pilha - ID 7
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
0
20000
40000
60000
80000
100000
120000

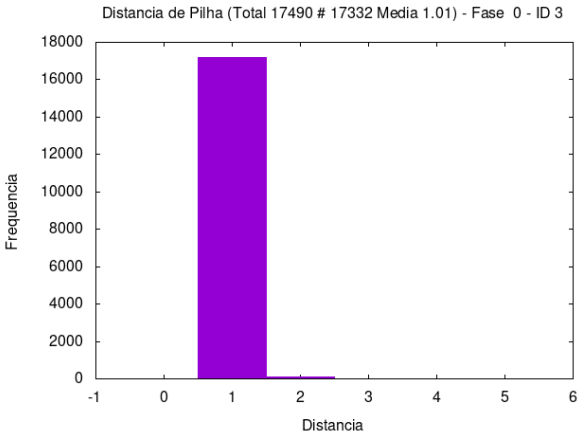
Da mesma forma em que foi possível notar quando que o acesso a cada dado da palavra, os gráficos de evolução de pilha também mostram esse fato. Outro ponto que é possível notar é o de que em geral acessos tendem a não serem superiores a 7 posições, isso pode se derivar do fato de que a grande maioria das manipulações feitas pelo QuickSort seguem a mediana passada e resultam em um uso de certa forma padronizado. Outra analise cabível de ser realizada é uma junção das conclusões chegas no acesso da memória, podemos ver que como temos os acessos sendo feitos sequencialmente não há grande distancias nas quais é possível acessar conteúdo e o conteúdo em geral rodeia sobre as cartas ele está geralmente em poucas posições atrás na pilha.

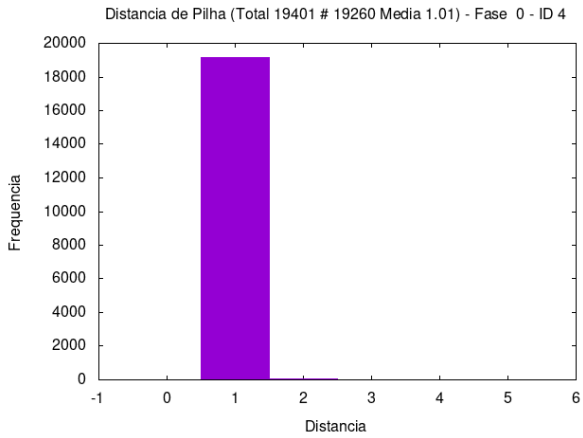
6.1.3 Distância De Pilha

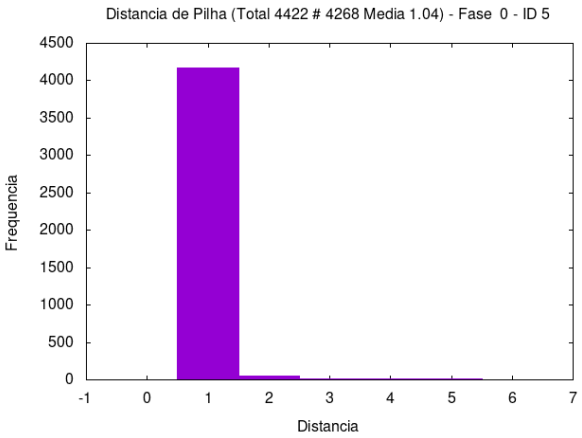


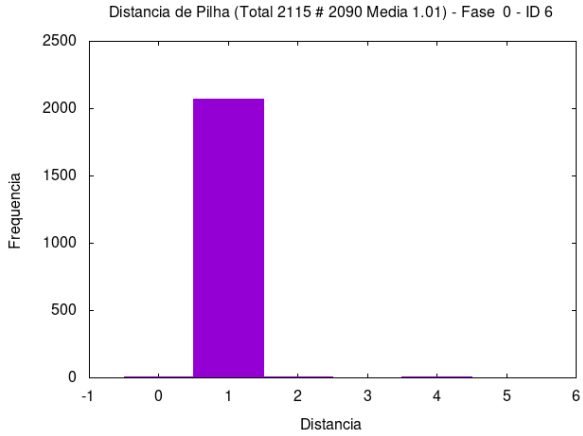


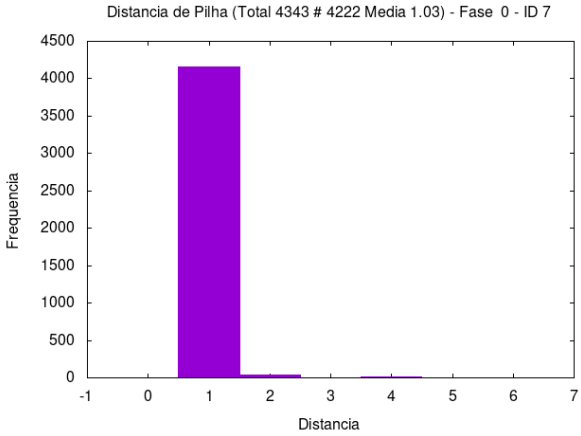






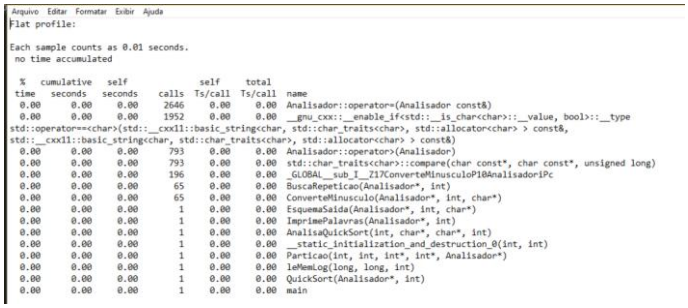






Analisando a distância de pilha as conclusões chegadas na evolução se tornam mais claras. Podemos notas que a frequência é focada na posição de distância 1, o que pode ser concluído desse fato é que em geral os dados que são necessários serem acessados estão a 1 “acesso” de distância, isso devido ao fator que a ordenação. Desse fato temos um menor consumo de memória e tempo de execução, já que, os dados foram alocados sequencialmente, basta apenas acessar o conteúdo guardado na posicao seguinte.

6.1.4 Gprof



```
Arquivo  Editar  Formatar  Exibir  Ajuda
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00     2646     0.00     0.00  Analisador::operator=(Analisador const&)
  0.00      0.00     0.00     1952     0.00     0.00  __gnu_cxx::__enable_if<std::__is_char<char>::__value, bool>::__type
std::operator==<char>(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&)
  0.00      0.00     0.00      793     0.00     0.00  Analisador::operator>(Analisador)
  0.00      0.00     0.00      793     0.00     0.00  std::char_traits<char>::compare(char const*, char const*, unsigned long)
  0.00      0.00     0.00      196     0.00     0.00  _GLOBAL__sub_I__Z17ConverteMinusculoP10AnalisadoriPc
  0.00      0.00     0.00       65     0.00     0.00  BuscaRepeticao(Analisador*, int)
  0.00      0.00     0.00       65     0.00     0.00  ConverteMinusculo(Analisador*, int, char*)
  0.00      0.00     0.00        1     0.00     0.00  EsquemaSaida(Analisador*, int, char*)
  0.00      0.00     0.00        1     0.00     0.00  ImprimePalavras(Analisador*, int)
  0.00      0.00     0.00        1     0.00     0.00  AnalisaQuickSort(int, char*, char*, int)
  0.00      0.00     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
  0.00      0.00     0.00        1     0.00     0.00  Particao(int, int, int*, int*, Analisador*)
  0.00      0.00     0.00        1     0.00     0.00  leMemLog(long, long, int)
  0.00      0.00     0.00        1     0.00     0.00  QuickSort(Analisador*, int)
  0.00      0.00     0.00        1     0.00     0.00  main
```

Ao realizar a analise gprof obtive todos os resultados gerados, uma das possíveis causas para esse resultado pode derivar do tempo de execução do programa que não consta na análise, outro motivo pode ser que a análise foi configurada de forma errada porém, outro caso pode ser o de que não é possível se extrair nenhuma expressão de tempo devido ao baixo tempo de execução. Existem diversos fatores que podem ser resultantes dessa impossibilidade de analise, porém não é possível afirmar com certeza qual deles é o culpado por esse infortuno. Então da análise gprof não é possível inferir nada.

6. Conclusão

Após a implementação do programa foi notório que ele pode ser visto de 4 momentos diferentes. A primeira foi de realizar as leituras e alocações corretamente dos atributos lidos do arquivo e distribui-los pela variáveis corretamente. A segunda é a de ser capaz de realizar corretamente a ordenação dos dados seguindo a nova ordem lexicográfica. O terceiro é realizar a contagem das repetições. O quarto e final é de pegar o resultado dessa ordenação e contagem e imprimi-lo num arquivo.

O grande ponto central deste trabalho era o de ser capaz de implementar funções e realizar estes procedimento de forma simples e objetiva, e que ao final da execução resultasse numa saída correta. Com um enfoque maior na implementação do QuickSort, que deve ordenar as palavras seguindo os critérios definidos anteriormente. O uso de técnicas de estruturas de dados foi imprescindível para a execução deste trabalho, pois sem ela a implementação seria catastroficamente complexa e resultaria num programa pouco funcional. Pode-se notar que o custo para executar o programa não foi tão alto, parte deste custo computacional não ser elevado se deve ao fato que pouquíssimas funções possuem o custo alto linear ou fixo, dado que em geral o custo é $O(n)$.

Portando é notório que a boa implementação e domínio das estruturas de dados, e a análise de custo são essenciais para que obtenhamos um programa o mais próximo do ideal, que realize o que é necessário de maneira requisitada, possua um custo computacional o mais baixo possível e seja simples de se entender. Requisitos os quais foram implementados no TP2.

7. Bibliografia

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

8. Instruções para compilação e execução

A compilação do código é feita através do Makefile, para a complicação usa-se o make que compila o programa e o executa, o comando make clean exclui todos os .o, executáveis e saídas geradas pelo programa.

Para se poder executar o programa é necessário haver uma pasta no mesmo diretório em que o programa se encontra nomeadas entrada e saída, sendo que inicialmente deve haver arquivos somente na primeira. Arquivos estes na extensão txt, que contenham os dados de entrada no formato especificado. Logo após adicioná-los a essa pasta basta alterar a linha de comando responsável por ele no Makefile e executá-lo que as saídas serão geradas no pasta saída.

Para executar a análise de desempenho basta apenas adicionar o comando -l ao no momento da execução