Trabalho Prático 2: Ordenador de Frases

Pedro de Oliveira Guedes
2021040008

DCC/UFMG

Introdução

A documentação aqui presente se refere a um programa produzido na linguagem de programação C + +, com o objetivo de implementar métodos de ordenação para palavras presentes em uma frase ou texto informado pelo usuário, de acordo com uma nova ordem lexicográfica, também informada pelo usuário.

O programa é estruturado para funcionar na seguinte ordem:

1. Leitura da nova ordem lexicográfica do arquivo informado pelo usuário;
2. Leitura do texto/frase do arquivo informado pelo usuário;
3. Ordenação das palavras lidas nas frases do arquivo de acordo com a nova ordem lexicográfica, também informada no arquivo;
4. Registro das palavras ordenadas num arquivo de saída.

Mais detalhes sobre a implementação do jogo serão dados no decorrer deste documento.

Método

Nesta seção, serão dadas informações sobre a organização do programa em pastas, bem como o que cada uma abriga, formatos de entrada e saída, configurações do sistema de testes, etc..

Configurações do sistema de testes

O programa foi implementado na linguagem C + +, compilado pelo G + +, compilador da GNU Compiler Collection. Além disso, foi utilizado o sistema operacional Ubuntu 9.3.0-17 ubuntu1~20.04 em um Windows 10, através do Windows Subsystem for Linux (WSL 2), com um processador AMD Ryzen 3 3200G (3.60GHz, 4 CPUs) e 13 GB RAM.

Formato de entrada e saída

O formato de entrada dos dados é através de um arquivo de texto, do formato que seja, seguindo o padrão:

```
#ORDEM
A Y   Z S    Q C  ...
```

```
#TEXTO
aquI TEmos Um   TEXTO!!!
AQUI também!
.
.
.
```

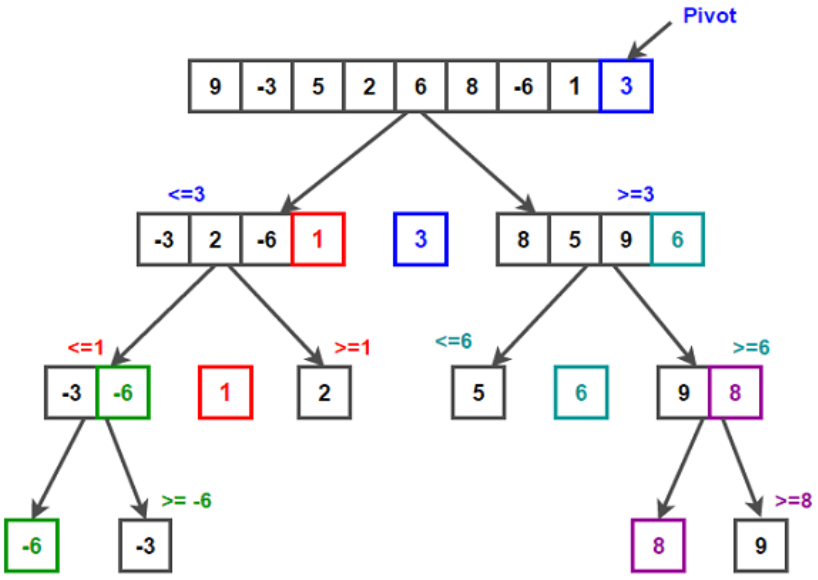
O formato de saída do programa também é através de um arquivo de texto, do formato que for informado pelo usuário, que segue o padrão:

```
<primeira palavra> <quantidade de ocorrências no texto do usuário>
<segunda palavra> <quantidade de ocorrências no texto do usuário>
<terceira palavra> <quantidade de ocorrências no texto do usuário>
.
.
.
```

Algoritmos e Estruturas de Dados

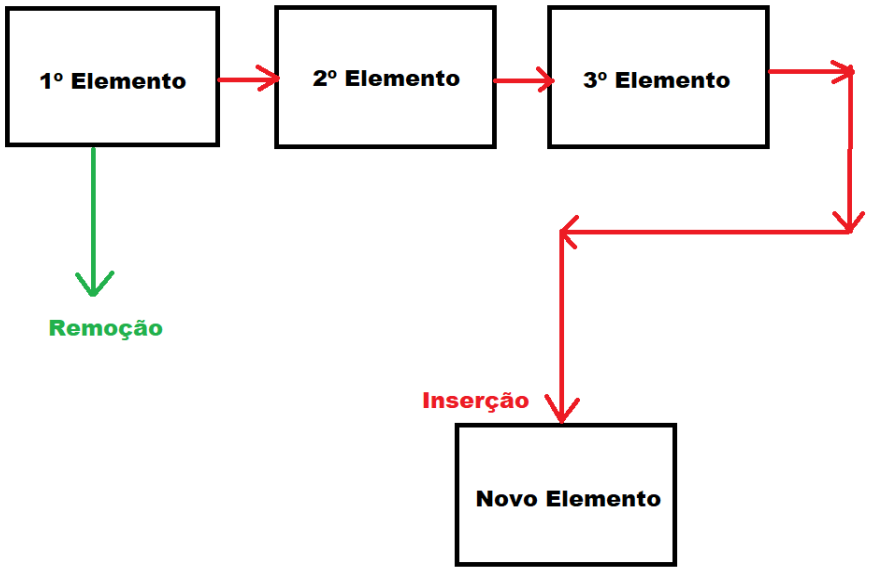
O principal algoritmo implementado foi o de *Quicksort,* tendo algumas adaptações para o problema, como comparação de elementos de acordo com a ordem lexicográfica informada pelo usuário e otimizações, como a mediana de "*n*" elementos para escolha do pivô e a ordenação por inserção das sub partições de até "*m*" elementos obtidos.

Este algoritmo está baseado na divisão do *array* inicial de elementos, escolhendo um pivô e posicionando os elementos maiores e menores à esquerda e à direita deste pivô, respectivamente. Podemos ver um exemplo de funcionamento deste algoritmo na imagem abaixo:



Além das estruturas de dados específicas para armazenar dados do problema, como a classe "*Word*" e "*LexOrder*", a Estrutura de Dados (ED) mais abrangente e geral implementada foi a de Lista Encadeada. Essencialmente, essa ED permite a inserção ilimitada de elementos, mantendo rastro de todos eles, o que é bastante útil nesse problema, já que não se sabe a quantidade de palavras do texto do usuário.

A utilização dessa lista encadeada pelo programa foi, na verdade, bastante similar à de uma fila, podendo ser substituída por ela, já que todas as inserções eram feitas ao final e remoções do começo, assim como ilustra a imagem a seguir:



Organização de pastas

A pasta raiz do projeto se chama "*TP*" e abriga, além das subpastas diretamente relacionadas ao programa, a subpasta "analisamem", que não será discutida neste documento, já que foi criada pelos professores da disciplina.

Além desta, a pasta também abriga as subpastas a seguir:

1. **bin:** Abriga todos os arquivos binários gerados pelo programa.
2. **include:** Inclui todos os arquivos de cabeçalho utilizados na implementação. Entre eles: *lex_order.hpp*, *memlog.hpp*, *msgassert.hpp*, *phrase_sorter.hpp*, *word_list.hpp*, *word.hpp*.
3. **obj:** Arquivos de objeto gerados para a compilação do programa principal.
4. **src:** Todos os arquivos de código fonte para a implementação do programa. Sendo eles: *lex_order.cpp*, *main.cpp*, *memlog.cpp*, *phrase_sorter.cpp*, *word_list.cpp*, *word.cpp*.

Análise de complexidade

A análise de complexidade contemplará as principais funções implementadas nos arquivos de código fonte do programa. Para todas as funções, serão consideradas operações relevantes:

- Complexidade de tempo: leitura e atribuição de valores, comparações.
- Complexidade de espaço: alocação de memória dinâmica, recebimento de estruturas dinâmicas e chamadas de funções.

A presente análise de complexidade leva em consideração a notação de complexidade "*grande O*". A título de referência, a quantidade de palavras informadas no arquivo de texto será denotada por "*n*", a quantidade de caracteres em cada palavra será denotada por "*c*", podendo ter subscritos (e.x.: c_1 ou c_2) quando houver mais de uma palavra envolvida.

Além disso, a quantidade de elementos considerados para a mediana de escolha do pivô será denotada por "*m*" e o tamanho da subpartição a ser ordenada por seleção por "*s*".

LexOrder(std::string lexOrder):

Esta função constrói um novo objeto do tipo LexOrder. Através da string recebida, que terá sempre 26 letras com espaços entre elas, ocorre uma iteração atribuindo valores às letras, ou seja, definindo a nova ordem lexicográfica, uma única vez durante o programa. Portanto, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(1)$.

LexOrder::getLetterValue(int letter):

Esta função recebe o código *ascii* de um caractere e a pesquisa entre as 26 letras da classe, retornando o valor atribuído à ela, ou o próprio código *ascii* do caractere pesquisado, caso ele não pertença ao alfabeto. Portanto, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(1)$.

*Word::isLessThan(Word *word, LexOrder *lexOrder)*:

Esta função itera por todos os "*c*" caracteres da menor palavra (*this* ou *word*) realizando comparações com a maior delas de acordo com a nova ordem lexicográfica definida, retornando verdadeiro, caso a palavra cujo método foi chamado (*this*) for menor que a recebida como parâmetro (*word*) e vice-versa. Sendo c_1 o tamanho de *this* e c_2 o tamanho

de *word*, dizemos que a complexidade de tempo é: $O(\ min(c_1,\ c_2)\ )$ e complexidade de espaço: $O(1)$.

*Word::isGreaterThan(Word *word, LexOrder *lexOrder)*:

Segue exatamente o mesmo princípio do método mencionado acima, com a única diferença de que o retorno é verdadeiro para quando *this* é maior que *word* de acordo com a ordem lexicográfica recebida. Ou seja, temos complexidade de tempo: $O(\ min(c_1,\ c_2)\ )$ e complexidade de espaço: $O(1)$.

WordList::push(std::string word):

Este método é responsável por inserir novos elementos na lista encadeada de palavras. Normalmente, a complexidade de tempo dele seria $O(1)$. Porém, como temos a restrição de não compreender palavras repetidas na lista, mas somar a quantidade de vezes que elas se repetem em um único elemento representante presente na lista, devemos comparar a palavra a ser inserida com todas as "*n*" palavras já presentes na lista. Dessa forma, temos complexidade de tempo: $O(n)$ e complexidade de espaço: $O(n)$.

WordList::popFromStart():

Este método remove sempre o primeiro elemento da lista encadeada. Portanto, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(n)$.

PhraseSorter::toLower(std::string str):

Este método é responsável por retornar a string recebida em casa baixa. Para isso, ele itera pelos "*c*" caracteres da string realizando conversões se necessário. Portanto, temos complexidade de tempo: $O(c)$ e de espaço: $O(1)$.

PhraseSorter::PhraseSorter(std::string inputFile, int medianSize, int insertionSize):

O construtor da classe PhraseSorter recebe o caminho de um arquivo com as entradas do programa, para cada uma das palavras, chama o método "*toLower*" e insere as palavras na

lista encadeada (como estudamos anteriormente, com custo $O(n)$) e, posteriormente, no array de palavras. Dessa forma, temos complexidade de tempo: $O(n^2 * c)$ e de espaço: $O(n)$.

PhraseSorter::insertionSort(int esq, int dir):

Este é um algoritmo de ordenação por inserção adaptado para as necessidades do problema. Ele itera pelos elementos a partir da posição informada pelo parâmetro "*esq*" até a posição informada por "*dir*" realizando comparações de acordo com a nova ordem lexicográfica. Portanto, temos complexidade de tempo: $O(\ (dir - esq)^2 * min(c_1, c_2)\ )$ e de espaço: $O(n)$, já que, apesar da ordenação ocorrer de "*esq*" até "*dir*", o *array* completo ainda está disponível para o método.

PhraseSorter::quickSort():

Este método faz a chamada dos métodos "*sort*" e "*qsPartition*" que, juntos, compõem o algoritmo de *Quicksort*. Em cada comparação realizada por este algoritmo, ele deve acessar caractere por caractere das palavras, utilizando a ordem lexicográfica fornecida. Dessa forma, apesar das otimizações e heurísticas implementadas, esse algoritmo ainda tem seu pior caso quando o array está invertido, apresentando complexidade de tempo: $O(n^2 * c)$ e de espaço: $O(n)$.

PhraseSorter::print(std::string outputFile):

Este método faz, primeiramente, a chamada do método "*quickSort*", que foi estudado logo acima. Após isso, itera por todas as "*n*" palavras armazenadas no *array* imprimindo-as no arquivo de saída uma a uma. Dessa forma, é conhecido que ele possui complexidade de tempo: $O(n^2 * c)$ e de espaço: $O(n)$.

Programa completo:

O programa completo faz a chamada basicamente de duas funções, o construtor da classe PhraseSorter e o método "*print()*", da mesma classe. Como já foi estudado anteriormente as complexidades de cada uma, podemos afirmar que a complexidade de tempo esperada é: $O(n^2)$ e a de espaço é: $O(n)$.

Obs:

As outras funções não serão analisadas quanto à complexidade porque se tratam de métodos que não são particularmente relevantes para o programa.

Estratégias de Robustez

A robustez em um programa é de crucial importância para o bom funcionamento do mesmo. Nesse sentido, uma série de verificações de entrada do usuário se fazem necessárias, para que não haja execuções errôneas ou estouros grosseiros de erro no programa. Para isso, foi utilizada a biblioteca *msgassert.h*, também presente nos arquivos deste trabalho.

As estratégias serão listadas e explicadas com relação à relevância de utilização.

- **Valores inválidos na linha de comando:** Dois dos parâmetros que o usuário pode fornecer para o programa durante sua execução são a quantidade de elementos relevantes para o cálculo da mediana e o tamanho da subpartição que será ordenada por inserção. Dessa forma, é importante que esses valores não sejam vazios, nem mesmo nulos. Portanto, há uma verificação sobre essa condição. Caso ela não seja atingida, o programa é finalizado e uma mensagem de erro é exibida para o usuário.
- **Falha na abertura de arquivos:** Caso algum arquivo informado pelo usuário não exista ou falhe ao ser aberto pelo programa, ele é finalizado e uma mensagem de erro explicando o ocorrido é exibida ao usuário.

Testes

Os testes para o programa implementado serão divididos entre testes de acesso à memória e testes de desempenho computacional para as diferentes operações. Os resultados das partidas serão armazenados em arquivos texto, assim como os tempos de execução. Mapas de acesso à memória em arquivos de imagem ".png" e o teste *gprof* também em um arquivo de texto.

O teste de memória verifica a performance do programa quanto o acesso à memória alocada pelo mesmo. Será utilizada para metrificar, a ferramenta da "distância de pilha". Basicamente, quando um endereço de memória é acessado, ele é inserido na pilha, quando ele for acessado novamente, ele é retirado da posição da pilha em que volta para a base da

mesma. A distância da posição em que o elemento estava ao ser chamado para a primeira posição da fila será a distância de pilha daquela chamada em questão.

A distância de pilha total será calculada pela média ponderada da frequência em que um elemento é desempilhado com aquela distância, sendo que a distância em si é o peso da média. Quanto menor for a distância de pilha, melhor projetado é o código, indicando menor custo de memória.

Os testes de desempenho verificarão o tempo que leva para que uma função seja executada para um determinado tamanho de entrada. Para isso, será marcado o tempo de início da execução do programa e o tempo de finalização do mesmo. A diferença entre estes tempos é então calculada e será o tempo de execução do programa.

Será analisado não somente o tempo individual de execução para diferentes entradas, mas a evolução deste tempo e se ele comprova ou não a análise de complexidade feita em um dos tópicos anteriores.

Análise Experimental

A análise experimental foi dividida entre testes de acesso à memória e testes de desempenho computacional, que serão abordados em tópicos abaixo. Para mais informações sobre os métodos de teste utilizados para esta análise experimental, consulte o tópico “Testes” deste documento.

Desempenho Computacional

Os testes de desempenho computacional foram realizados utilizando textos de domínio público da internet, produzidos por Machado de Assis. Serão realizados 5 testes de desempenho, sendo que a cada novo teste, será acrescentada uma nova obra, das que se encontram a seguir, exatamente nessa ordem:

1. Memórias Póstumas de Brás Cubas (292.991 caracteres sem espaço)
2. Esaú e Jacó (339.780 caracteres sem espaço)
3. Dom Casmurro (305.394 caracteres sem espaço)
4. Quincas Borba (368.268 caracteres sem espaço)
5. Helena (272.348 caracteres sem espaço)

Ou seja, o primeiro arquivo terá 292.991 caracteres com a obra “Memórias Póstumas de Brás Cubas”, o segundo terá 632.771 caracteres, com as obras “Memórias Póstumas de

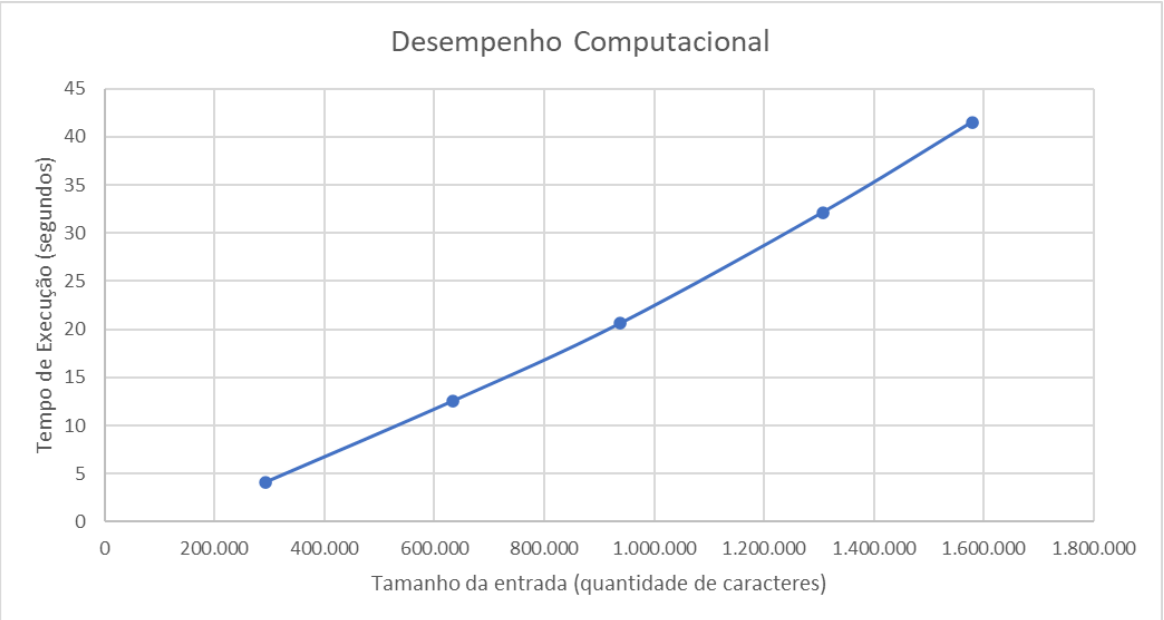
Brás Cubas” e “Esaú e Jacó”. Todos os arquivos terão exatamente a mesma ordem lexicográfica (alfabeto invertido Z-A).

A seguir, será apresentado o resultado obtido para este teste.

Evolução do Desempenho Computacional

Veremos uma tabela que mostra a evolução do desempenho com o crescimento da quantidade de rodadas a serem computadas pelo programa. Será exibido também um gráfico que mostra esta evolução de forma mais visual.

Quantidade de caracteres e nome do arquivo	Tempo inicial	Tempo final	Duração da execução
292.991 entrada_1.out	4137.586226243	4141.711990587	4.125764344
632.771 entrada_2.out	4141.713501776	4154.241171841	12.527670065
938.165 entrada_3.out	4154.242597061	4174.855941431	20.613344370
1.306.433 entrada_4.out	4174.857265638	4206.987112658	32.129847020
1.578.781 entrada_5.out	4206.988547016	4244.709493967	41.519568176



A complexidade de tempo do programa total havia sido estimada como sendo $O(n^2)$ ignorando a quantidade de caracteres por palavra, já que é algo difícil de se mensurar e representar numa equação. Porém, é válido ressaltar que essa é a complexidade de pior caso do programa e é refletida, principalmente, pelo algoritmo de *Quicksort*.

As entradas para o teste não foram projetadas para testar o pior caso, mas simplesmente o desempenho do programa como um todo. Sendo assim, o correto é observar a complexidade de caso médio do programa, que também é fortemente influenciada pelo caso médio do algoritmo de *Quicksort*. Este, conhecidamente, tem complexidade de caso médio e melhor caso como $O(n * log(n))$.

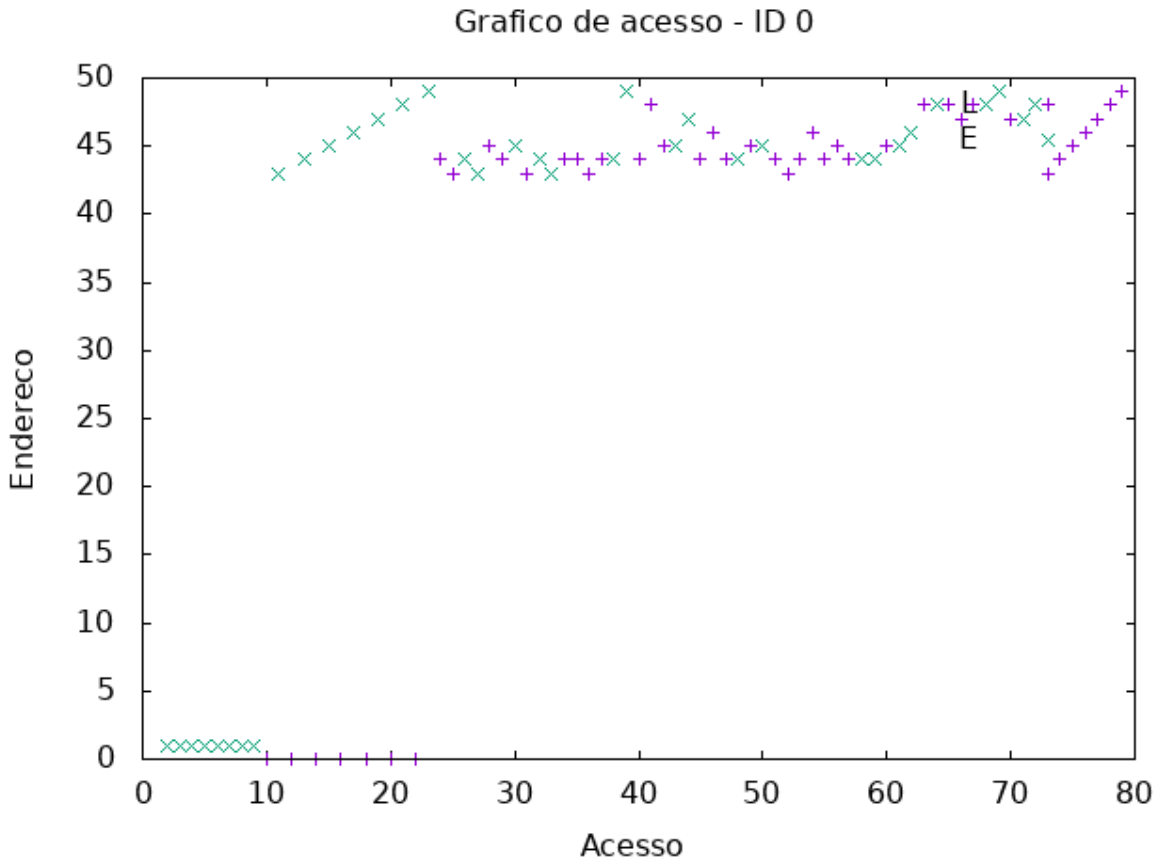
Observando o gráfico exibido acima, percebemos que é uma representação bastante similar à de um gráfico de $f(n) = n * log(n)$. Ou seja, através desse experimento, foi comprovado que os estudos de complexidade feitos são válidos, já que $n * log(n) < n^2$.

Análise de Padrão de Acesso à Memória e Localidade de Referência

Os testes de memória serão realizados com uma entrada de 23 caracteres, para melhor visualização de acesso. Não serão repetidos estes testes, já que eles são apenas para geração dos gráficos.

Mapa de acesso à memória

Este gráfico mostra como os endereços de memória do programa são acessados a cada chamada de função. Pontos em formato de “X” na cor verde representam acesso à memória para escrita, já os em formato de “+” na cor rosa, representam acesso para leitura.



Podemos observar que há uma grande amplitude entre os endereços de memória acessados. Isso se deve ao fato de que, inicialmente, as palavras são inseridas numa lista encadeada, sendo, posteriormente, inseridas em um array, que foi alocado somente depois da leitura de todas as palavras da entrada.

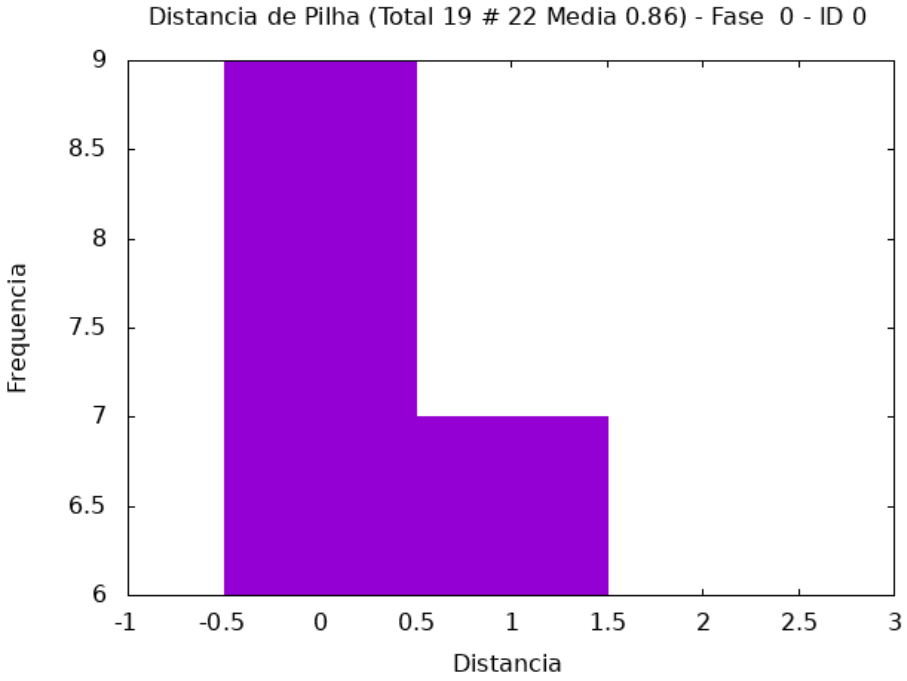
Os primeiros acessos (1 a 23) são provenientes da leitura do arquivo de input do programa, gravando os dados na lista encadeada e, logo após, os inserindo no array. Os próximos acessos (24 a 72) são para ordenação do array, através do algoritmo de *Quicksort*. Os acessos restantes são de acesso ao array para impressão das informações nele armazenadas..

Distância de pilha

Estes gráficos são chamados de histogramas e mostram a distância de pilha dos acessos aos endereços de memória alocados para as estruturas de dados implementadas.

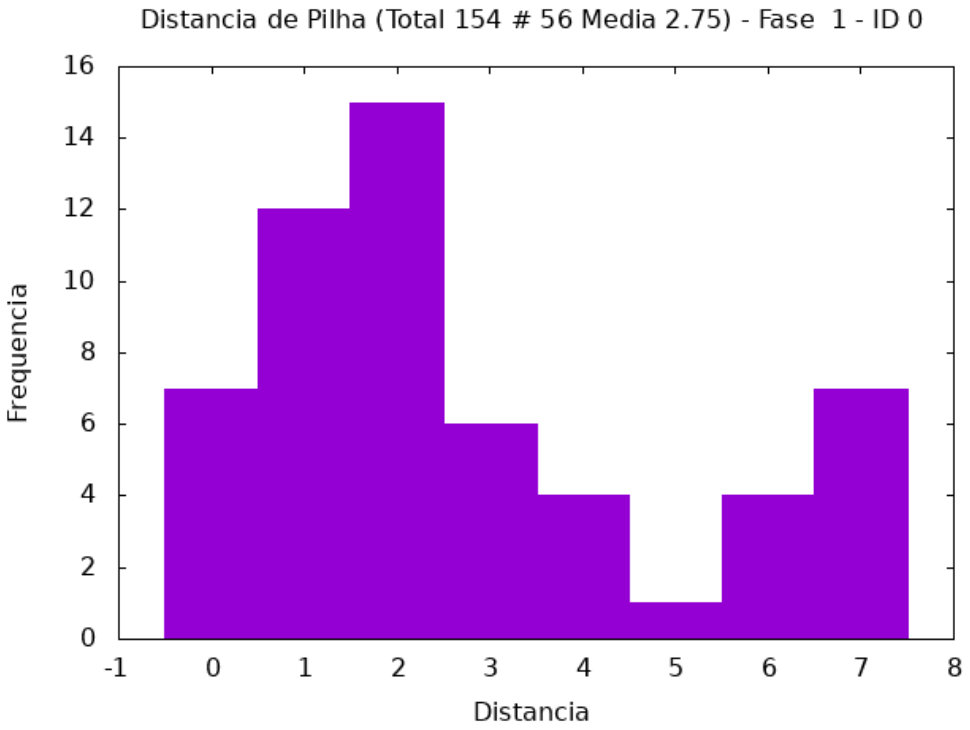
Durante a fase 0, o programa está responsável pela leitura dos dados do arquivo de entrada, bem como a inserção das palavras lidas numa lista encadeada e a transferência delas

para um *array* dinamicamente alocado. Podemos visualizar as distâncias de pilha desse trabalho na imagem a seguir:



A frequência da distância de pilha igual a zero representa a quantidade de palavras lidas no texto de entrada, já que todas elas passam pelo processo de inserção na lista encadeada, ainda que representem repetições. Já a frequência da distância de pilha igual a um, representa a quantidade de palavras não repetidas na entrada, que serão utilizadas ao decorrer do programa para ordenação e impressão do resultado final.

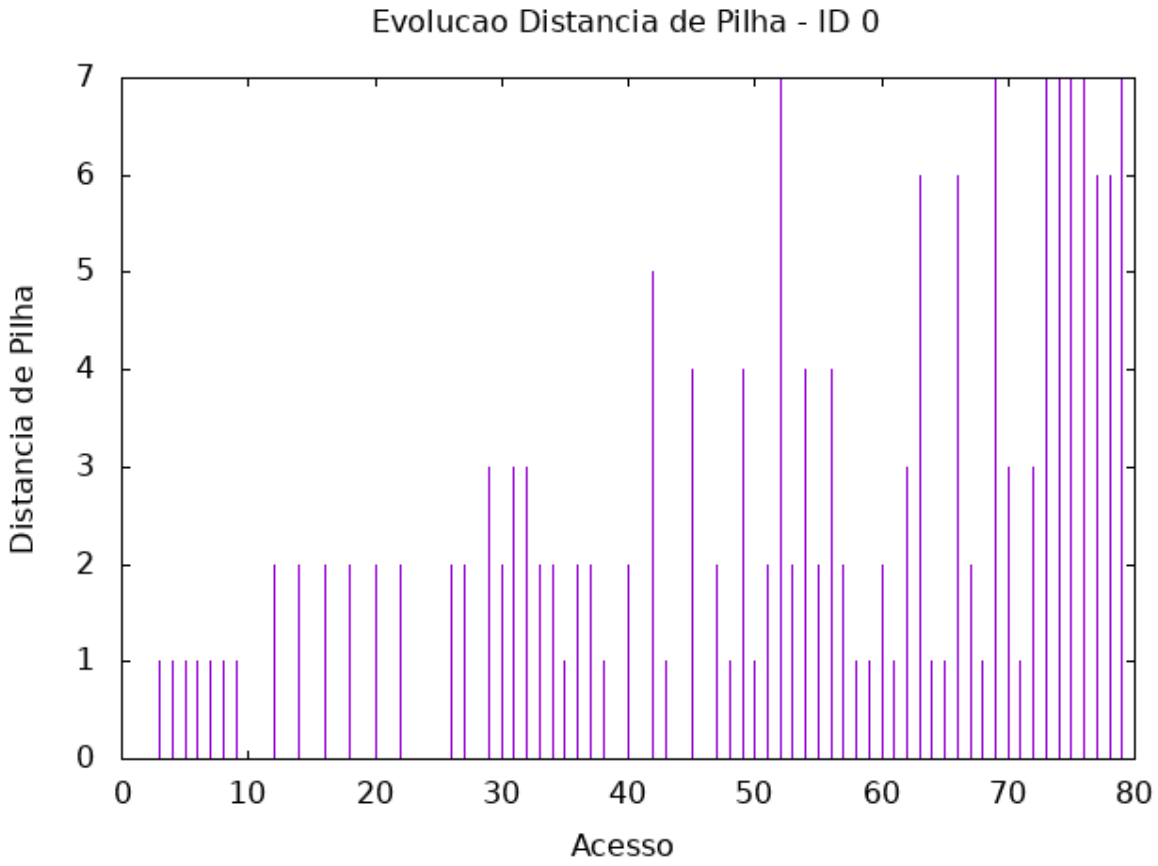
Já na fase 1, o programa está responsável pela ordenação do *array* de palavras alocado, bem como a impressão do mesmo. Portanto, espera-se que as distâncias de pilha sejam de no máximo 7, que é a quantidade total de palavras do *array*, já que é o que foi observado no gráfico visto acima. Podemos visualizar o acesso à memória para a ordenação do *array* por *Quicksort* e suas respectivas otimizações e heurísticas no histograma a seguir:



Evolução da Distância de Pilha

A distância de pilha deve ser mais fraca e repetitiva nos primeiros acessos, já que eles se referem à leitura das palavras para a lista encadeada e cópia para o *array*. Já o final deve possuir acessos mais frequentes e intensos, apesar de não haver uma boa previsibilidade quanto à distância de pilha de cada um dos acessos, já que o posicionamento das palavras no *array* não é necessariamente conhecido.

Por fim, a evolução da distância de pilha deve obedecer à condição da máxima distância de pilha ser a quantidade de palavras no *array*.



Conclusão

Neste trabalho, foram implementados alguns Tipos Abstratos de Dados (TADs) para armazenar mais informações sobre uma palavra, como a quantidade de vezes que ela aparece em um texto, assim como outros TADs menos específicos, como uma lista encadeada. Além disso, foram implementados os algoritmos de ordenação *insertion sort* e *Quicksort*, necessários para a ordenação das palavras lidas.

Para que as operações de ordenação citadas no parágrafo anterior fossem possíveis, também foi necessário criar um TAD para a nova ordem lexicográfica informada pelo usuário do programa no arquivo de entrada. Através dessa nova ordem lexicográfica, foram implementadas novas formas de comparação de palavras, de acordo com a posição das letras e tamanho total das mesmas, de forma a tornar a ordenação por qualquer algoritmo possível, e compatível com as especificações do usuário.

Toda a implementação foi feita na linguagem C + +, utilizando o compilador G + +, sendo também desenvolvido em um subsistema de Linux para Windows (WSL 2). Foi possível executar os conceitos vistos em sala de aula sobre estruturas de dados e algoritmos

de ordenação, bem como realizar a análise de complexidade de cada um destes e exercitar as boas práticas de programação em C + +.

O trabalho contribuiu bastante para a consolidação dos conhecimentos teóricos de estruturas de dados apresentados na disciplina, sendo especialmente importante para a visualização de aplicações práticas dos conhecimentos adquiridos.

Bibliografia

Meira, W. and Pappa, G. (2021). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Visão geral do Quicksort. Trabalhos para Escola, 2018. Disponível em: <https://trabalhosparaescola.com.br/visao-geral-do-quicksort/>. Acesso em: 02 de julho de 2022.

Instruções para compilação e execução

1. Extraia o arquivo .zip presente;
2. Abra um terminal na pasta raiz do mesmo;
3. Digite “make”. Isso compilará o programa;
4. Digite “./bin/tp2.exe <argumentos>”.

    Os argumentos podem ser:

    a. -i / -I <input.txt> (arquivo com as frases a serem ordenadas)
    b. -o / -O <output.txt> (arquivo para registrar as frases ordenadas)
    c. -m / -M <numero_inteiro> (quantidade de elementos a serem considerados na mediana para escolha do pivô. 5 por padrão)
    d. -s / -S <numero_inteiro> (Tamanho da subpartição a ser ordenada por inserção. 20 por padrão)
    e. -p <arquivo_de_logs.txt> (arquivo para registrar os resultados computacionais de performance e/ou memória da partida)
    f. -l Caso esta flag esteja presente, o acesso à memória é registrado no arquivo de logs de “-p”

OBS: O programa também pode ser executado sem parâmetro algum. Ele está configurado para, por padrão, acessar como input o arquivo "in.txt" e como output o arquivo "out.txt”, desde que ambos estejam na mesma pasta em que o terminal foi aberto, ou seja, a pasta raiz do projeto.

É possível também executar o comando “*./bin/tp2.exe -h*”. Com isso, as instruções de utilização serão exibidas na tela.