Trabalho Prático 2

Carlos Henrique Brito Malta Leão

Matrícula: 2021039794

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brasil

chbmleao@ufmg.br

1. Introdução

O objetivo desse trabalho é realizar uma análise do número de ocorrências das palavras usadas em um texto baseada em uma nova ordem lexicográfica. O programa desenvolvido nesse trabalho é capaz de ler todas as palavras contidas em um arquivo de texto de entrada e relatar no arquivo de saída todas as palavras que ocorrem no texto, seguido do número de vezes que essa palavra foi encontrada, considerando letras maiúsculas e minúsculas como as mesmas. Além disso, o arquivo de saída deve ser ordenado em relação a uma ordem específica, passada também no arquivo de entrada do programa.

Essa documentação tem como objetivo explicitar como foi realizada a implementação desse sistema, além de realizar uma análise explicativa sobre como o sistema do jogo funciona. Ademais, também serão explicitados a robustez, abstração e desempenho do programa, possibilitando um melhor entendimento de como o sistema funciona na prática.

Para a resolução do problema supracitado, foi criado um programa na linguagem C++, que utiliza três principais classes: *Node*, *List* e *Vector*. Essas três classes representam duas estruturas de dados muito importantes. Primeiramente, a classe *List*, representa uma lista de *strings* encadeada alocada dinamicamente, em que a classe *Node* representa as células dessa lista. Por fim, a classe *Vector* representa um vetor de *strings* alocado, também, dinamicamente. Ademais, temos a estrutura *memlog*, que auxilia no processo de análise de desempenho do programa e as funções *msgassert*, que são utilizadas de forma a aumentar a robustez do programa.

Por fim, ao decorrer dessa documentação, alguns aspectos sobre o trabalho serão melhor explicados, como a descrição da implementação na seção 2 e a análise de complexidade na seção 3. Em seguida, nas seções 4 e 5 serão explicitadas as estratégias de robustez e a análise experimental. Por fim, teremos a conclusão do trabalho, sumarizando o que foi aprendido durante seu desenvolvimento.

2. Método

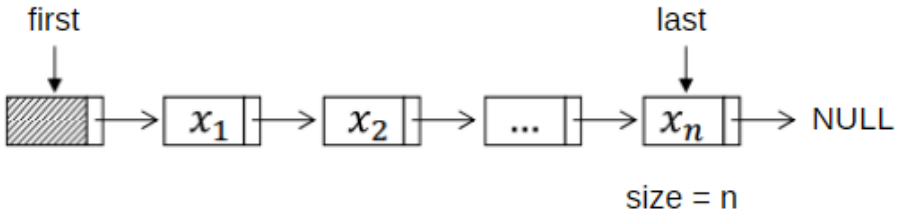
Durante essa seção, será realizada uma descrição da implementação do programa estruturada na linguagem C++. Dessa forma, serão detalhadas as estruturas de dados, classes e funções implementadas.

2.1 Lista encadeada

O programa desenvolvido apresenta a implementação de uma lista encadeada alocada dinamicamente, essa estrutura de dados é formada pelas classes *List* e *Node*. Essa estrutura de dados foi escolhida, já que, inicialmente durante a leitura do arquivo de entrada, não há como saber a quantidade de palavras que precisam ser

armazenadas, dessa forma, a melhor forma de armazenar cada uma dessas palavras é utilizando uma lista encadeada.

Na lista, as palavras são armazenadas em posições não contíguas da memória, dessa forma foi necessário a utilização de células que são encadeadas usando apontadores. Dessa forma, a classe *Node* representa as células da lista implementada na classe *List*. Além disso, como a lista utiliza alocação dinâmica, é possível aumentar e diminuir o tamanho da lista se necessário. Um diagrama esquemático da lista encadeada implementada pode ser visto na figura a seguir.



Nesse sentido, a classe *Node* é muito simples ela apresenta apenas dois atributos, sendo eles a *string* que será armazenada uma palavra e o endereço da próxima célula da lista, chamado "*next*", próximo. Além disso, essa classe apresenta algumas funções muito simples que serão explicadas em seguida.

- *Node()*: Inicializa a célula com uma *string* vazia e *next* nulo.
- *Node(string word)*: Inicializa a célula com a *string* recebida como parâmetro e *next* nulo.

Em seguida, podemos nos aprofundar mais na classe principal da lista encadeada, a classe *List*, que demandou uma maior complexidade em sua implementação. Essa classe apresenta quatro atributos, *size* armazena o tamanho atual da lista, *first* e *last* guardam a primeira e última células da lista respectivamente, e *lettersOrder* armazena a ordem lexicográfica que deve ser utilizada para a impressão do arquivo de saída ao fim da execução.

Por fim, esta classe apresenta algumas funções específicas, que abstraem e aumentam a robustez do código:

- *List():* Inicializa a lista com tamanho 0 e cria um nós que será tanto o atributo *first* quanto o atributo *last*, ou seja, essa célula será o começo e o fim da lista, já que a lista está vazia.
- *~List():* Chama a função clean e exclui a primeira célula da lista.
- *getSize():* Retorna o atributo de tamanho atual da lista.
- *isEmpty():* Retorna se a lista está vazia.
- *SetPositon(int pos):* Cria um nó auxiliar p, com o endereço da primeira posição, que será iterado por toda a lista encadeada. Essa iteração será realizada utilizando um loop que será iterado o número da posição que deve ser retornado, todas as vezes atribuindo para p o endereço do próximo item da lista. Ao final, o nó p é retornado pela função. Em alguns processos, é necessário receber o nó anterior ao desejado, então é possível retornar esse nó também.
- *getWord(int pos):* Retorna a palavra localizada na posição indicada utilizando a função *setPosition().*

- *insertAtStart(string word):* Insere o item na primeira posição. Primeiramente, uma “nova célula” é criada e inicializada com o endereço do objeto jogador que deseja inserir na lista. Dessa forma, o atributo “próximo” da primeira célula é atribuído ao “próximo” da “nova célula”. Em seguida, o “próximo” da primeira célula é atribuído o endereço na “nova célula”. Por fim, o tamanho da lista é incrementado. Caso o “próximo” da “nova célula” for um endereço nulo, significa que a fila só possui um elemento, então o novo final da fila é atribuído para a “nova célula”.
- *insertAtEnd(string word):* Insere o item na última posição. Primeiramente, uma “nova célula” é criada e inicializada com o endereço do objeto jogador que deseja inserir na lista. Dessa forma, no atributo “próximo” da última célula é atribuído o endereço da “nova célula”. Em seguida, o endereço da última posição da célula é mudado para o endereço da “nova célula”. Por fim, o tamanho da lista é incrementado.
- *insertAtPosition(string word, int pos):* Insere o item na posição indicada. Primeiramente, é preciso usar a função *setPosition* para posicionar na posição anterior a desejada e atribuir a célula retornada a variável “p”. Em seguida, o endereço do objeto do jogador é atribuído a “nova célula”, e o seu “próximo” apontado para o “próximo” da célula “p”. Por fim, o “próximo” de “p” é apontado para a “nova célula” e o tamanho da lista é incrementado. Caso o “próximo” da “nova célula” seja um endereço nulo, é preciso apontar o endereço “último” para a “nova célula”.
- *removeAtStart():* Remove o item da primeira posição. Primeiramente, o endereço da célula a ser removida é armazenado na variável “p”, então o endereço do “próximo” da primeira célula é apontado para o “próximo” de “p” e o tamanho da fila é incrementado. Caso o “próximo” da primeira célula seja nulo, então o último elemento também é o primeiro elemento. Por fim, a célula “p” é excluída e o item da célula excluída é retornado.
- *removeAtEnd():* Remove o item da última posição. Primeiramente, diferente da remoção ao início, é preciso utilizar a função *setPosition* para encontrar o elemento anterior ao último “p”. Em seguida, é atribuído nulo ao “próximo” de “p” e o tamanho da lista decrementado. Por fim, o último elemento é excluído, o endereço do último da lista é apontado para p e o item da célula excluída é retornado.
- *removeAtPosition():* Remove o item da posição indicada. Primeiramente, assim como a remoção da última célula, é preciso usar a função *setPosition* para posicionar “p” na posição anterior a célula que deseja remover. Em seguida, fazemos com que a célula auxiliar “q” aponte para o “próximo” de “p” e o próximo de “p” aponte para o “próximo” de “q”, dessa forma, a célula alvo já foi retirada da lista. Ao final, basta excluir a célula “q” e verificar se o “próximo” endereço de “p” é nulo, caso seja, é preciso atribuir o endereço do último termo da lista para o endereço de “p”. Por fim, o item da célula excluída é retornado.
- *search(string name):* Retorna o item com o nome igual ao que está sendo procurado. A busca é realizada passando por todos os valores da lista até que ela acabe. Caso encontre um jogador com o nome recebido como o parâmetro, a função termina e ele é retornado, caso contrário a célula passa para a próxima da lista. Caso o loop chegue ao final da lista e não encontre nenhum jogador com o nome igual ao recebido como parâmetro, um jogador com atributos inválidos é retornado.
- *setLettersOrder(order):* Armazena cada caractere da ordem lexicográfica apenas em letras minúsculas.

- *print():* Função utilizada exclusivamente para o processo de depuração do código.
- *printOrder():* Imprime a ordem lexicográfica que será utilizada no arquivo de saída.
- *clean():* Função que limpa a lista encadeada, excluindo todas as suas células.
- *setPosition(int pos):* Função privada auxiliar que participa do processo de inserção e remoção de itens.
- *search(string word):* Retorna o item com o nome igual ao que está sendo procurado.
- *clean():* Função que limpa a lista encadeada, excluindo todas as suas células.
- *setPosition(int pos):* Função privada auxiliar que participa do processo de inserção e remoção de itens.
- *passListToVector():* Passa todos os elementos da lista encadeada para um vetor que também armazena strings. Além disso, também é passada a ordem lexicográfica. Retorna o vetor criado.

2.2 Vetor

O vetor foi implementado utilizando alocação estática, recebendo um tamanho máximo. Dessa forma, essa estrutura permite acesso aleatório a qualquer posição em tempo $O(1)$, além de permitir percorrer a lista em ambas direções caso necessário, começando na posição 0 e terminando na posição $n-1$, em que n representa o tamanho do vetor. Nesse sentido, essas vantagens dessa estrutura de dados tornam a utilização do algoritmo de ordenação Quick Sort, que será melhor explicado posteriormente, muito mais eficiente.

Além disso, esta classe apresenta alguns atributos: size armazena o tamanho do vetor; items armazena um arranjo de palavras; lettersOrder armazena a ordem lexicográfica; medianArraySize armazena a quantidade de itens que devem ser ordenados para obter um pivô a cada chamada recursiva do algoritmo Quick Sort; e arrayMinimumSize armazena o tamanho da partição que deve ser utilizado um algoritmo de ordenação mais simples, no caso, o algoritmo de inserção. Por fim, essa classe apresenta algumas funções específicas, que abstraem e aumentam a robustez do código:

- *Vector():* Inicializa o objeto vetor com atributos inválidos.
- *Vector(int size):* Inicializa o objeto vetor com tamanho igual ao recebido como parâmetro e aloca o espaço na memória para os itens que serão inseridos no vetor.
- *~Vector():* Destrutor da classe, atribui valor inválido para o tamanho do vetor.
- *accessVector():* Função utilizada exclusivamente para o processo de análise de desempenho do programa, acessando todos os itens alocados no vetor.
- *writeElement(T item):* Armazena um item na próxima posição do vetor que ainda não foi armazenada, e soma 1 ao atributo position, para armazenar a posição do próximo item a ser inserido.

- *readElement(int pos):* Retorna o elemento da posição desejada acessando a posição do item direto no vetor.
- *print():* Função utilizada exclusivamente no processo de desenvolvimento e depuração do código. Ela imprime cada palavra armazenada no arranjo de strings.
- *printOrder():* Imprime a ordem lexicográfica que será utilizada no arquivo de saída.
- *printOutFile(ofstream &outputfile):* Itera todas as palavras do vetor, como o vetor já está ordenado, caso seja encontrada a mesma palavra encontrada anteriormente, o contador é aumentado. Nesse sentido, ao encontrar uma palavra diferente, a antiga palavra é impressa com seu contador, e o processo começa novamente.
- *setLettersOrder(char lettersOrder[])*: Armazena a ordem lexicográfica recebida como parâmetro.
- *getCharValue(char c)*: Retorna o valor de um caractere de acordo com a ordem lexicográfica armazenada. Por exemplo, se a ordem é "Z, Y, X, ..., C, B, A" e a letra C é recebida como parâmetro, a função retornará o valor 23.
- *biggerWord(string word1, word2)*: Retorna 1 se a palavra um for "maior" que a palavra 2, ou seja, deve aparecer antes da palavra 2. Caso a segunda palavra deva aparecer primeiro, retorna 2. Por fim, caso ambas as palavras sejam iguais, a função retorna 0. Nesse sentido, essa função utiliza a função *getCharValue*, para encontrar o valor lexicográfico de cada uma das palavras.
- *quickSort(int medianArraySize, arrayMinimumSize)*: Chama a função auxQuickSort(), passando zero e o tamanho do vetor subtraído um como parâmetros. Além disso, atribui o tamanho do arranjo que é necessário utilizar sua mediana como pivô no algoritmo de ordenação Quick Sort. Por fim, também é atribuído o tamanho mínimo da partição do vetor que não é mais eficiente utilizar o algoritmo Quick Sort, em que será utilizado o algoritmo Insertion Sort.
- *auxQuickSort(int left, right)*: Essa função trata o algoritmo Quick Sort de forma recursiva, em que a ideia básica é dividir o problema de ordenar um conjunto com n itens em problemas menores que são ordenados independentemente e que são combinados ao final. Primeiramente, o pivô é escolhido ao encontrar a mediana dos m primeiros termos do vetor, valor m é recebido como parâmetro durante a execução do programa.

  Em seguida, o vetor é percorrido com um índice i a partir da esquerda, até que o i-ésimo termo seja maior ou igual ao pivô. Da mesma forma, o vetor é percorrido com índice j a partir da direita, até que o j-ésimo termo seja menor ou igual ao pivô (todas essas comparações em relação à ordem lexicográfica recebida como parâmetro no arquivo de entrada). Em seguida, os dois elementos encontrados são trocados de posição. Esse processo continua até os apontadores i e j se cruzarem.

  Por fim, caso j seja maior que o índice da esquerda, a função é chamada recursivamente, agora ordenando o vetor começando no índice da esquerda até j. Além disso, caso i seja menor que o índice da direita, a função também é chamada recursivamente, agora ordenando o vetor que começa no índice i e termina no índice da direita. Dessa forma, o processo se repete até que o tamanho do vetor a ser ordenado seja igual ao parâmetro s recebido como parâmetro na execução do programa. Quando o vetor tem esse tamanho ou menor, a função *insertionSort* é

chamado, já que o algoritmo de inserção é mais eficiente que o Quick Sort em arranjos de menor tamanho. Esse algoritmo será melhor explicado a seguir.

- *insertionSort(int left, right)*: O algoritmo de inserção itera sobre cada elemento do vetor da esquerda para a direita e, à medida que avança, ordena os elementos à esquerda. Isso ocorre da seguinte forma: consiste em cada passo a partir do segundo elemento selecionar o próximo item da sequência e colocá-lo em seu local apropriado de acordo com a ordem lexicográfica recebida. Dessa forma, a cada elemento iterado, é realizado um loop com todos os elementos à sua esquerda, até encontrar um elemento que seja "maior" que ele, ou seja, deva aparecer depois que ele na impressão do arquivo de saída.

2.3 Main

O arquivo main apresenta duas partes principais que são representadas pelos dois blocos do arquivo de entrada "#ORDEM" e "#TEXTO". Para realizar cada um dos processos, é executado um loop duas vezes, uma vez para cada um dos possíveis blocos supracitados.

Primeiramente, caso o bloco de ordem lexicográfica seja encontrado, o programa chama a função *runOrder*. Nesse sentido, a função lê toda uma linha do arquivo e chama a função da lista *setLettersOrder*, assim, a lista armazena a ordem lexicográfica para a impressão ao final.

Em seguida, caso o bloco de leitura do texto seja encontrado, o programa chama a função *runText*. Nesse sentido, enquanto o arquivo tiver conteúdo para ser lido, cada palavra encontrada é convertida para ter apenas letras minúsculas e inserida ao final da lista.

Por fim, ao acabar a leitura do arquivo, um vetor é criado pela função *passListToVector* e a lista encadeada é excluída para liberar memória. Então, o vetor é ordenado chamando a função *quickSort* e, ao final, o arquivo de saída é criado e escrito.

3. Análise de complexidade

Esta seção apresenta a análise de complexidade de tempo e espaço para a execução completa do programa, desde a leitura do arquivo até a impressão do arquivo de saída na ordem lexicográfica correspondente. Para um melhor entendimento, realizaremos a análise da complexidade de tempo primeiro e em seguida a análise de espaço.

3.1 Análise de Complexidade de Tempo

Para entender melhor a complexidade do programa, primeiramente realizaremos a análise de complexidade de cada função de forma individual, para, em seguida, analisar a complexidade do programa como um todo. Além disso, para uma melhor organização, analisaremos uma classe implementada de cada vez.

LIST

A classe *List* é uma lista encadeada que armazena n elementos, palavras. Dessa forma, utilizaremos a quantidade de elementos n para realizar a análise de complexidade. Ademais, algumas funções tratam cada caractere da palavra de forma separada, assim, trataremos a quantidade de letras como m.

- *List():* Inicializa os atributos da classe *List*, operações constantes com tempo

$O(1)$.

- *~List():* Destrutor da classe que chama a função *clean*, $O(n)$ e exclui a primeira célula da lista, $O(1)$. Dessa forma, a complexidade assintótica da função será: $O(1) + O(n) = O(\max(1, n)) = O(n)$.
- *getSize():* Apenas retorna um atributo, operação constante com tempo $O(1)$.
- *isEmpty():* Apenas realiza uma comparação, operação constante com tempo $O(1)$.
- *setPosition():* Realiza uma iteração até chegar na posição desejada, dessa forma, o melhor caso é $O(1)$, caso procure o primeiro elemento, e o pior caso é $O(n)$, caso procure o último elemento da lista. Dessa forma, teremos a complexidade assintótica da função como: Melhor caso: $O(1)$ e Pior caso: $O(n)$.
- *getWord():* Inicializa um objeto da classe *Node*, $O(1)$, e chama a função *setPosition*, melhor caso $O(1)$ e pior caso $O(n)$. Dessa forma, a complexidade assintótica da função terá pior caso ($O(1) + O(n) = O(\max(1, n)) = O(n)$) e melhor caso ($O(1) + O(1) = O(1)$).
- *insertAtStart():* Realiza operações constantes em tempo $O(1)$, apenas mudando o endereço de variáveis. Dessa forma, a complexidade assintótica da função é $O(1)$.
- *insertAtEnd():* Realiza operações constantes em tempo $O(1)$, apenas mudando o endereço de variáveis. Dessa forma, a complexidade assintótica da função é $O(1)$.
- *insertAtPostion():* Chama a função *setPosition* com melhor caso $O(1)$ e pior caso $O(n)$. Além disso, realiza operações constantes em tempo $O(1)$, apenas mudando o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) = O(1)$.
- *removeAtStart():* Realiza operações constantes em tempo $O(1)$, apenas mudando e excluindo o endereço de variáveis. Dessa forma, a complexidade assintótica é $O(1)$.
- *removeAtEnd():* Chama a função *setPosition* com melhor caso $O(1)$ e pior caso $O(n)$. Além disso, realiza operações constantes em tempo $O(1)$, apenas mudando e excluindo o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) = O(1)$.
- *removeAtPosition():* Chama a função *setPosition* com melhor caso $O(1)$ e pior caso $O(n)$. Além disso, realiza operações constantes em tempo $O(1)$, apenas mudando e excluindo o endereço de algumas variáveis. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) = O(1)$.
- *search():* Realiza operações constantes em tempo $O(1)$. Além disso, chama a função *setName* e *isEmpty*, ambas $O(1)$. Por fim, realiza um loop que itera os elementos da lista até encontrar o elemento desejado, melhor caso $O(1)$ e pior caso $O(n)$. Dessa forma, teremos a complexidade assintótica da função como Pior caso: $O(1) + O(1) + O(1) + O(n) = O(\max(1, n)) = O(n)$ e Melhor caso: $O(1) + O(1) + O(1) + O(1) = O(1)$.
- *setLettersOrder:* Realiza operações constantes em tempo $O(1)$. Além disso, realiza um loop que itera sempre 26 vezes, a quantidade de letras do alfabeto. Dessa forma, teremos a complexidade assintótica da função como $26 * O(1) = O(1)$.

- *print():* Realiza operações constantes em tempo $O(1)$. Além disso, realiza um loop que itera todos os elementos da lista, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$.
- *printOrder():* Realiza operações constantes em tempo $O(1)$. Além disso, realiza um loop que itera sempre 26 vezes, a quantidade de letras do alfabeto. Dessa forma, teremos a complexidade assintótica da função como $26 * O(1) = O(1)$.
- *clean():* Realiza operações constantes em tempo $O(1)$. Além disso, realiza um loop que itera todos os elementos da lista, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$.
- *passListToVector():* Realiza operações constantes em tempo $O(1)$. Além disso, realiza um loop que itera todos os elementos da lista, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$.

VECTOR

A classe *Vector* é um vetor que armazena n elementos, palavras. Dessa forma, utilizaremos a quantidade de elementos n para realizar a análise de complexidade.

- *Vector():* Inicializa todos os atributos da classe *Vector*, $O(1)$.
- *~Vector():* Realiza operações constantes com tempo $O(1)$.
- *accessVector():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza uma iteração que percorre todos os elementos do vetor, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$
- *writeElement():* Realiza operações constantes com tempo $O(1)$.
- *readElement():* Realiza operações constantes com tempo $O(1)$.
- *print():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza uma iteração que percorre todos os elementos do vetor, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$.
- *printOrder():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza um loop que itera 26 vezes, quantidade de letras do alfabeto. Dessa forma, a complexidade assintótica da função é $26 * O(1) = O(1)$.
- *printOutFile():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza uma iteração que percorre todos os elementos do vetor, $O(n)$. Dessa forma, teremos a complexidade assintótica da função como $O(1) + O(n) = O(\max(1, n)) = O(n)$.
- *setLettersOrder():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza um loop que itera 26 vezes, quantidade de letras do alfabeto. Dessa forma, a complexidade assintótica da função é $26 * O(1) = O(1)$.
- *getCharValue():* Realiza operações constantes com tempo $O(1)$. Além disso, realiza um loop que itera 26 vezes, quantidade de letras do alfabeto. Dessa forma, a complexidade assintótica da função é $26 * O(1) = O(1)$.
- *biggerWord():* Realiza operações constantes com tempo $O(1)$. Além disso realiza um loop que, no melhor caso, itera apenas uma vez, $O(1)$, caso as primeiras letras de cada palavra sejam diferentes. No pior caso, é preciso percorrer $m - 1$ caracteres para encontrar o caractere que é diferente, $O(m)$. Nesse loop, a função *getCharValue* é chamada duas vezes, $2 * O(1)$. Dessa forma, a complexidade assintótica da função no melhor caso é $O(1) + O(1) * \big(2 * O(1)\big) = O(1)$, já no pior

caso é $O(1) + O(m) * \big(2 * O(1)\big) = O(m)$.

- *auxQuickSort():* Essa função é um pouco mais complexa, dessa forma, calcularemos o melhor e pior caso de forma separada:
  - Melhor caso: Ocorre quando o vetor é dividido em 2 partes iguais em todas as suas partições. Dessa forma, a função realiza $O(n)$ operações para cada partição e realiza $O(\log n)$ partições. Dessa forma, como a função é recursiva, temos a seguinte equação de recorrência:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

  Que resulta na seguinte ordem assintótica:

$$C(n) = O(n \log n)$$

  - Pior caso: Ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos do vetor, isto faz com que o procedimento seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada, o que resultaria em uma ordem de complexidade $O(n^2)$. Porém, como foi implementada uma melhoria no algoritmo, em que o pivô é escolhido a partir de uma mediana de x elementos do vetor, o pior caso é evitado, já que torna-se impossível utilizar o vetor como sendo um dos extremos do vetor.
  - Além disso, o algoritmo também chama a função insertionSort, que tem melhor caso $O(n)$ e pior caso $O(n^2)$, mas como essa função só é chamada para lidar com partições do arranjo muito pequenas, o custo dessa função é praticamente linear, o que melhor a eficiência do algoritmo com relação a essas partições pequenas.
  - Dessa forma, podemos determinar a complexidade assintótica do algoritmo Quick Sort como $O(n \log n)$, que representa o melhor caso e o caso médio determinado por Sedgewick e Flajolet em 1999.
  - Por fim, cada comparação realiza precisa de chamar a função biggerWord, com tempo $O(m)$, em que m representa o número de caracteres das palavras escolhidas. Dessa forma, a complexidade assintótica final da função *auxQuickSort* é $O(mn \log n)$.
- *quickSort():* Realiza operações constantes com tempo $O(1)$. Além disso, chama a função auxQuickSort com tempo $O(mn \log n)$. Dessa forma, a complexidade assintótica da função é $O(1) + O(n \log n) = O(\max(1,\ mn \log n)) = O(mn \log n)$.
- *insertionSort()*: Realiza operações com tempo $O(1)$. Além disso, realiza dois loops, em que, o loop externo sempre será executado n vezes. Já o loop interno, no melhor caso, ele será executado apenas uma vez, com tempo $O(1)$, que ocorre quando o vetor já está ordenado. Já no pior caso, o loop interno será executado i vezes, em que i representa a i-ésima iteração do loop externo, com tempo $O(n^2)$.
- *swap():* Realiza operações constantes com tempo $O(1)$.

MAIN

Agora que já analisamos todas as funções de todas as classes, torna-se mais simples o processo de análise da complexidade assintótica da função *Main*. Durante toda a execução do programa, a função *main* realiza operações constantes com tempo $O(1)$, como a abertura de arquivos, criação e atribuição de variáveis. Dessa forma, podemos partir para as duas principais funções do arquivo *main*.

A função *runOrder* realiza operações constantes com tempo $O(1)$. Além disso, a função *setLettersOrder* é chamada, também com tempo $O(1)$. Dessa forma, a complexidade assintótica dessa função é $O(1) + O(1) = O(1)$.

A função *runText* também realiza operações constantes com tempo $O(1)$. Além disso, realiza uma iteração por todas as n palavras do arquivo, com tempo $O(n)$. Além disso. A cada iteração é chamada a função convertStringToLower, que tem tempo $O(m)$, já que é preciso iterar os m caracteres da palavra passada como parâmetro. Dessa forma, a complexidade final dessa função é $O(mn)$.

Por fim, mais algumas funções específicas da lista e do vetor são chamadas. São elas: *passListToVector*, com tempo $O(n)$, *quickSort*, com tempo $O(mn \log n)$ e *printOutFile*, com tempo $O(n)$.

Em conclusão, podemos sumarizar todas as ordens de complexidade que encontramos para encontrar a seguinte ordem de complexidade para o programa desenvolvido, em que n representa o número de palavras no bloco de texto do arquivo de entrada, e m representa a quantidade de caracteres por palavra.

$$O(1) + O(1) + O(mn) + O(n) + O(mn\ log\ n) + O(n) = O(\max(1, mn, n, mn \log n))$$

$$\boldsymbol{O(mn \log n)}$$

3.2 Análise de Complexidade de Espaço

Para realiza a análise de complexidade de espaço, precisamos analisar a função *main* como um todo, já que ela lida com toda a execução do programa. Nesse sentido, a função *main* apresenta três principais partes, as funções *runOrder* e *runText*, e as linhas finais do arquivo, que lidam com a criação do vetor e a sua ordenação.

Primeiramente, a função *runOrder* trata o preenchimento do arranjo que contém a ordem lexicográfica, dessa forma, a função lida com um arranjo de 26 caracteres. Dessa forma, a complexidade de espaço dessa função pode ser definida da seguinte forma:

$$26O(1) = O(1)$$

Em segundo lugar, a função *runText* trata o preenchimento da lista encadeada, assim, essa função lida com as n palavras que são escritas no bloco de texto do arquivo de entrada. Dessa forma, a função lê n strings e também aloca n strings na lista encadeada. Por fim, a complexidade de espaço dessa função pode ser definida da seguinte forma:

$$O(n)$$

Por fim, podemos analisar o restante da função *main*. Nesse sentido, essa seção trata de três principais funções, sendo elas *passListToVector*, *quickSort* e *printOutFile*. A função *passListToVector*, lida com os elementos da lista que já foram armazenados e também com os novos elementos que serão inseridos no vetor dinâmico, ou seja, a função lida com $2n$ elementos, com complexidade de espaço $2 * O(n) = O(n)$. Além disso, as funções quickSort e printOutFile lidam com apenas os n elementos, palavras, do vetor, ambas apresentando complexidade de espaço $O(n)$.

Em conclusão, podemos sumarizar todas as ordens de complexidade encontradas durante a execução do programa para encontrar a complexidade espacial do programa como um todo:

$$O(1) + O(n) + O(n) + O(n) + O(n)$$

$$= \boldsymbol{O(n)}$$

4. Estratégias de Robustez

Robustez é a capacidade de um sistema funcionar mesmo em condições anormais. Nesse sentido, as funções implementadas no programa criado apresentam verificações de entradas inadequadas e mal funcionamento do programa. Para tornar o código mais robusto e protegido, foram utilizadas funções da biblioteca *msgassert.h*, disponibilizada no *moodle* da matéria Estrutura de Dados.

Primeiramente, analisaremos a robustez das funções das duas estruturas de dados implementadas, lista encadeada e vetor. Em seguida, poderemos observar a robustez da função *main* como um todo.

4.1 Lista Encadeada

- *setPosition():* Verifica se a posição passada como parâmetro é maior ou igual a zero e menor que o tamanho da lista. Caso essa verificação seja falsa, a execução do programa é interrompida.
- *Todas as funções de remoção:* Verifica se a lista não está vazia. Caso essa verificação seja falsa, a execução do programa é interrompida.

4.2 Vetor

- *~Vector():* Verifica se o tamanho do vetor é maior que zero. Caso a verificação seja falsa, é impresso “Vector has already been destroyed” no terminal.
- *Vector():* Verifica se o tamanho do vetor passado como parâmetro é maior que zero. Caso a verificação seja falsa, a execução do programa é interrompida. Além disso, é verificado se o vetor *items* tem o endereço diferente de nulo. Caso contrário, a execução do programa é interrompida.
- *readElement():* Verifica se a posição recebida como parâmetro é maior ou igual a zero e menor que o tamanho do vetor. Caso contrário, a execução do programa é interrompida e é impresso “Invalid index” no terminal.
- *writeElement():* Verifica se o vetor não está cheio. Caso contrário, a execução do programa é interrompida e é impresso “Vector is already full” no terminal.

4.3 Main

- Verifica se o parâmetro ‘-i’, que representa o nome do arquivo de texto de entrada, foi passado durante a execução do programa. Caso contrário, o arquivo de entrada padrão ‘entrada.txt’, localizado na raiz do projeto, é utilizado, além de imprimir uma mensagem de aviso no terminal.
- Verifica se o parâmetro ‘-o’, que representa o nome do arquivo de texto de saída, foi passado durante a execução do programa. Caso contrário, o arquivo de saída padrão ‘saida.txt’, localizado na raiz do projeto, é utilizado, além de imprimir uma mensagem de aviso no terminal.
- Verifica se o parâmetro ‘-m’, que representa a escolha do pivô realizando uma mediana de m elementos, foi passado durante a execução e se m é maior ou igual a 1. Caso contrário, o número 1 é atribuído ao valor de m e é impresso uma mensagem de aviso no terminal.
- Verifica se o parâmetro ‘-s’, que representa o tamanho s da partição do vetor que deve ser utilizado um algoritmo de ordenação mais simples, foi passado durante a execução do programa e se s é maior ou igual a 1. Caso contrário, o número 1 é atribuído ao valor de s e é impresso uma mensagem de aviso no terminal.

- Verifica se o parâmetro ‘-p’, que representa o arquivo de registro de memória, foi passado durante a execução do programa. Caso contrário, o arquivo de registro de memória padrão ‘/tmp/tp2log.out’ é utilizado, além de imprimir uma mensagem de aviso no terminal.
- A função *convertStringToLower* verifica se a string recebida como parâmetro não está vazia. Caso contrário, a mensagem de aviso ‘Empty string’ é impressa no terminal.
- Verifica se foi possível acessar os arquivos de entrada e saída. Caso contrário, a execução do programa é interrompida e é impresso uma mensagem de aviso que indica qual arquivo não pôde ser acessado.

5. Análise Experimental

Esta seção apresenta os experimentos realizados em termos de desempenho computacional e localidade de referência. Os resultados desses experimentos foram gerados utilizando as bibliotecas disponibilizadas no *Moodle* da disciplina de estrutura de dados *analisamem* e *memlog*. Por fim, essa seção também apresenta uma análise desses resultados, para um melhor entendimento sobre a utilização de memória durante a execução do programa.

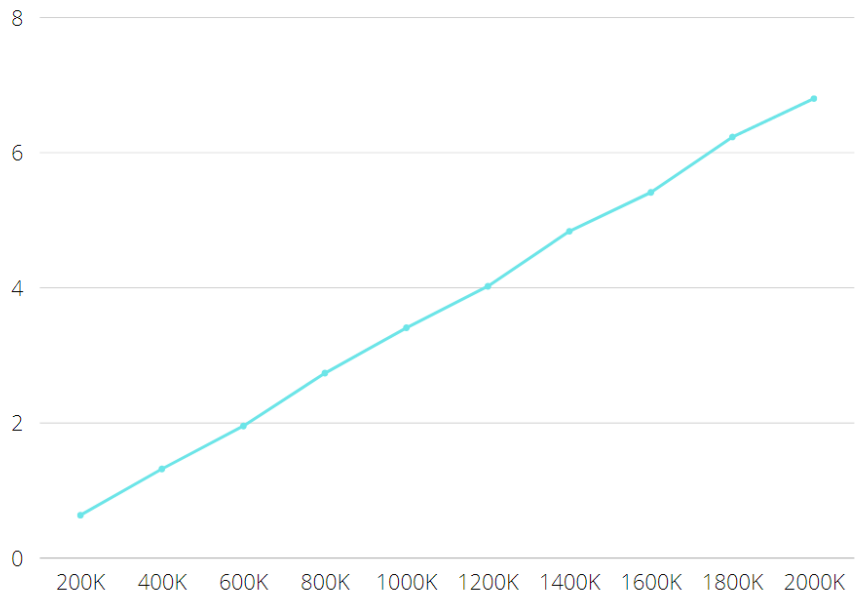
5.1 Análise de Desempenho

Primeiramente, realizaremos a análise de desempenho. Para isso, foram utilizados arquivos de entrada gerados por um programa simples na linguagem Python, que lê um arquivo principal de dois milhões de palavras, e dessa forma, faz as partições necessárias. Dessa forma, foi possível utilizar entradas variando o número total de palavras, mas mantendo sempre a mesma ordem de ordenação (ordem inversa da ordem alfabética).

Cada arquivo utilizado apresenta um número de palavras diferente, utilizando dez arquivos variando de 200 mil para 2 milhões de palavras. Além disso, em relação ao algoritmo de ordenação Quick Sort, a determinação do pivô foi a escolha da mediana dos três primeiros elementos, ademais, a partir de partições de tamanho 20, um algoritmo mais simples de ordenação foi utilizado, como já foi citado e explicado melhor nas seções anteriores. A seguir, temos a tabela comparando o tempo de execução de cada texto, além de um gráfico ilustrativo do incremento do tempo de execução em relação ao crescimento do número de palavras.

NÚMERO DE PALAVRAS	RESULTADO DE DESEMPENHO
200 mil	I 1 12923.867170100 F 2 12924.504407300 R 0.637237200
400 mil	I 1 12924.506236200 F 2 12925.828173000 R 1.321936800
600 mil	I 1 12925.830484900 F 2 12927.786794800 R 1.956309900
800 mil	I 1 12927.789755900 F 2 12930.528146500 R 2.738390600
1 milhão	I 1 12930.531611600 F 2 12933.941445300 R 3.409833700

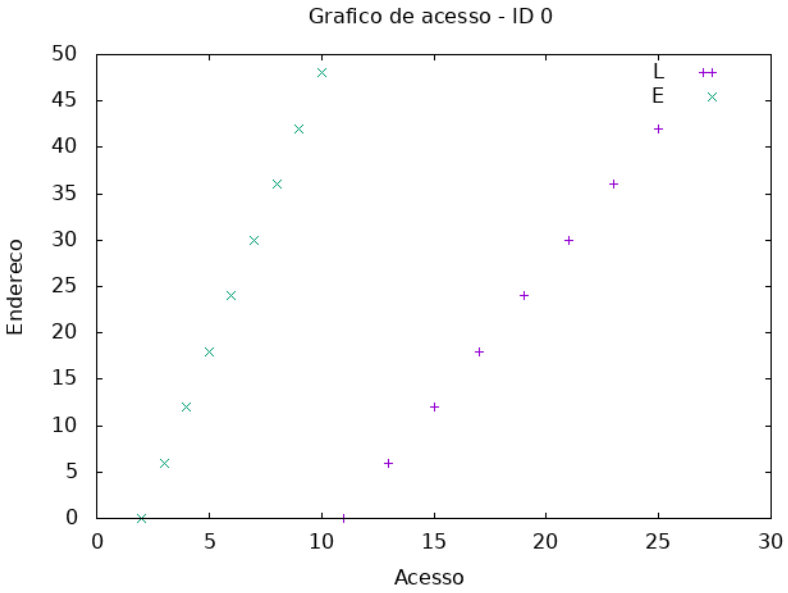
1 milhão e 200 mil	I 1 12933.945466000 F 2 12937.968559800 R 4.023093800
1 milhão e 400 mil	I 1 12937.973164500 F 2 12942.810694600 R 4.837530100
1 milhão e 600 mil	I 1 12942.815899000 F 2 12948.228779200 R 5.412880200
1 milhão e 800 mil	I 1 12948.234896300 F 2 12954.465930400 R 6.231034100
2 milhões	I 1 12954.472139900 F 2 12961.272597400 R 6.800457500



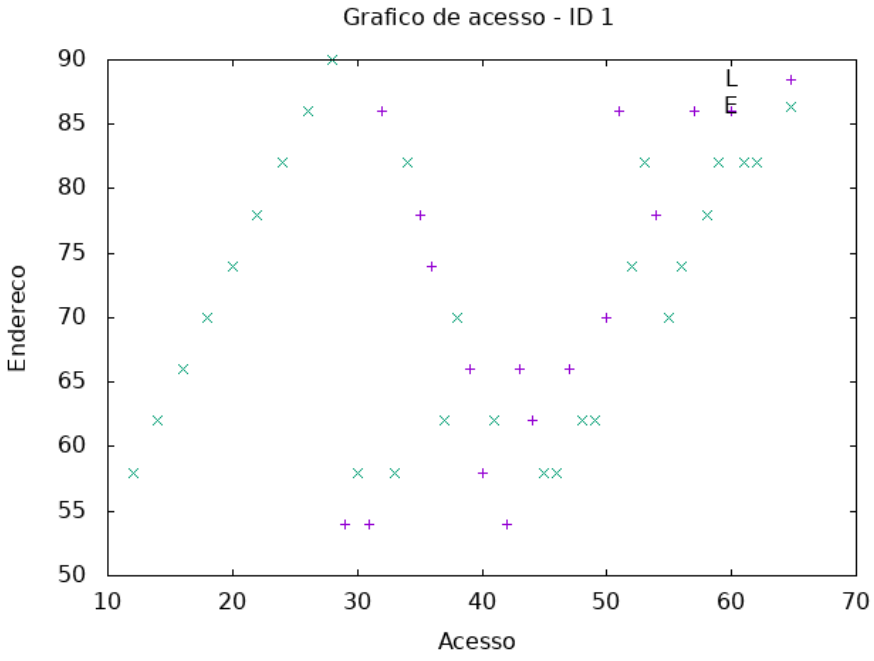
Analisando o gráfico, percebemos que ele apresenta um crescimento semelhante à um crescimento $n \log n$. Nesse sentido, esse resultado apresenta grande semelhança à complexidade assintótica encontrada durante a análise da seção 3 dessa documentação, em que foi determinada a complexidade de tempo do programa como $O(mn \log n)$, com n representando o número de palavras e m representando o número de letras de cada palavra. Como o número de letras de cada palavra são números arbitrariamente pequenos, podemos considerar m como uma constante, dessa forma, teremos a complexidade assintótica $O(n \log n)$, como pode ser observado no gráfico ilustrativo acima.

5.2 Localidade de Referência

Em seguida, para realizar a análise de localidade de referência, analisaremos especificamente cada palavra do bloco de texto do arquivo de entrada. Nesse sentido, o arquivo de entrada utilizado para realizar essa análise contém um texto de apenas nove palavras, para facilitar a visualização dos gráficos gerados. Nesses gráficos, o 'ID 0' representa a lista encadeada utilizada, enquanto o 'ID 1' representa o vetor. Primeiramente, analisaremos os gráficos de acesso, que apresentam o endereço de cada atributo em relação ao número de seu acesso.

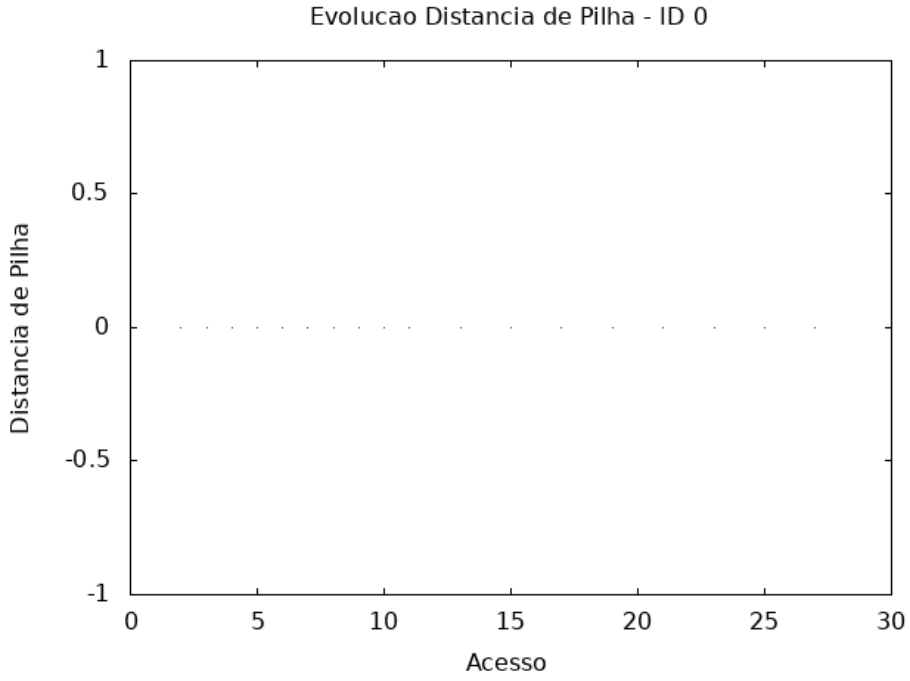


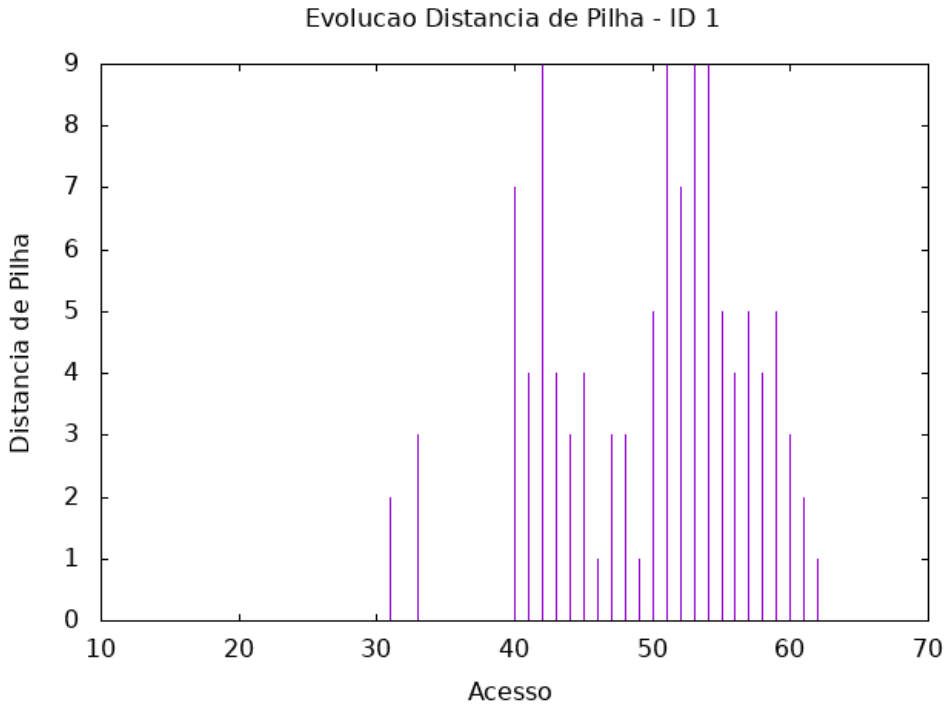
O gráfico acima, representa os acessos aos endereços da lista encadeada. Primeiramente, todos as palavras lidas no arquivo de entrada são escritas nos nós da lista, representada pelas marcações verdes no gráfico. Em seguida, todos os elementos são acessados para realizar a escrita do vetor de palavras, que pode ser visto no gráfico a seguir.



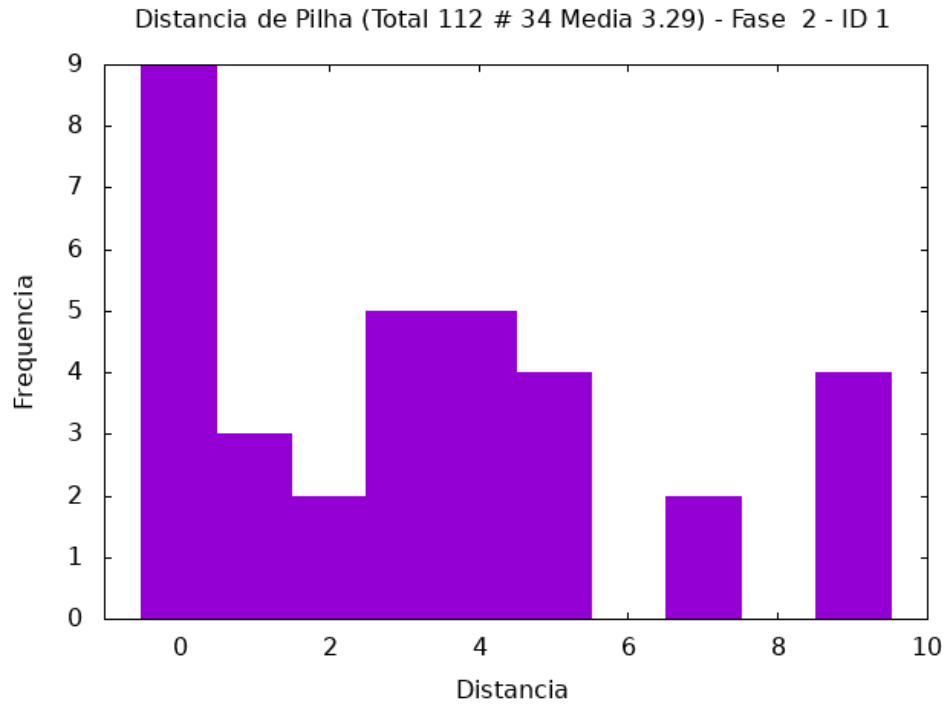
Esse gráfico inicia com a escrita de todos os elementos do vetor. Em seguida, as duas extremidades do vetor são acessadas, isso indica o início do algoritmo de ordenação Quick Sort. Dessa forma, os elementos são acessados iterando nas duas direções, até a condição de parada, quando as duas palavras são trocadas de lugar e escritas no endereço uma da outra. Assim, o algoritmo continua para partições menores, mas a visualização no gráfico torna-se um pouco mais difícil.

Em segundo lugar, podemos analisar os gráficos de distância de pilha e sua evolução.





Podemos ver no gráfico de ‘ID 0’, que representa a lista encadeada, sempre apresenta distância de pilha igual a zero. Já no gráfico de ‘ID 1’, que representa o vetor, apresenta distâncias de pilha que variam muito, isso ocorre devido ao algoritmo Quick Sort, que realiza partições no vetor de forma pouco previsível, já que essas partições variam de entrada para entrada, demonstrando a adaptabilidade do algoritmo de ordenação implementado. Isso pode ser melhor observado no seguinte gráfico de distância de pilha especificamente do algoritmo Quick Sort.



Por fim, concluímos que, mesmo que o algoritmo Quick Sort seja um dos algoritmos de ordenação mais eficientes, devido a sua grande adaptabilidade à entrada, sua distância de pilha varia muito, podendo chegar até o tamanho n total do vetor. Concluímos que mesmo apresentando uma distância de pilha inconstante, ele ainda é um ótimo e eficiente algoritmo de ordenação.

6. Conclusão

Este trabalho lidou com o problema de realizar uma análise do número de ocorrências de palavras em um texto, as ordenando um uma nova ordem lexicográfica, utilizando estruturas de dados e algoritmos de ordenação aprendidos durante o curso de Estrutura de Dados. Nesse sentido, a abordagem utilizada foi a criação de um programa na linguagem C++, que utiliza duas estruturas de dados, sendo elas a lista encadeada e o vetor dinâmico, além de algoritmos de ordenação, com foco principal no Quick Sort híbrido com o algoritmo de inserção.

Com a solução adotada, pode-se verificar que é possível analisar textos com um número arbitrariamente grande de palavras, como foi exemplificado durante os testes de performance na seção 5. Essa solução permitiu realizar uma análise sucinta de como a execução do programa ocorre em relação à memória do computador, tanto em relação ao tempo como em relação ao espaço, observando os gráficos e resultados de desempenho gerados pela execução do Makefile juntamente com as bibliotecas *analisamem* e *gprof*.

Por meio da resolução desse trabalho, foi possível praticas os conceitos relacionados às estruturas de dados trabalhados durante o curso, além de um entendimento muito mais profundo sobre o funcionamento do algoritmo de ordenação Quick Sort. Nesse sentido, a implementação desse algoritmo foi muito desafiadora, por se tratar de um algoritmo recursivo que precisava também da implementação de algumas melhorias. Como a implementação de uma melhor escolha do pivô, para evitar o pior caso, e a utilização do algoritmo Insertion Sort para menores partições de elementos. Em conclusão, todas essas implementações desafiadoras auxiliaram no entendimento aprofundado de algoritmos de ordenação, principalmente o Quick Sort.

7. Bibliografia

Slides virtuais e códigos da disciplina de estruturas de dados disponibilizados via moodle.

Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8. Instruções para Compilação e Execução

- Primeiramente, utilizando um terminal, acesse o diretório TP.
- Ao digitar o comando ***make all*****,** o comando *build* do arquivo Makefile é executado no terminal, e, dessa forma, os arquivos objetos serão gerados na pasta *obj*, enquanto o executável do programa será criado com nome 'tp2.exe' na pasta *bin*.
- Além disso, outros comandos também podem ser executados:
    - ***perf:*** Lê os arquivos "200kText.txt" até "2000kText.txt", executando o texto e a ordem lexicográfica de cada um, variando de 200 mil até 2 milhões de palavras. Esses arquivos são retirados da pasta *assets*. Além disso, os resultados de desempenho são armazenados, retornando o tempo de execução de cada texto. Os resultados de desempenho são armazenados na pasta *tmp*, para ver os resultados, basta acessar a pasta *tmp* no terminal e escrever comandos no seguinte formato: 'more 200kText.out'.
    - ***mem:*** Executa o arquivo "entrada.txt" localizado na raiz do diretório, de preferência um arquivo com poucos jogadores e um menor número de rodadas, armazenando no arquivo de saída "saída.txt". Além disso, os acessos a memória são registrados e podem ser utilizados para a plotagem de gráficos utilizando a biblioteca disponibilizada *analisamem*, juntamente com o *gnuplot*.
    - ***run:*** Executa o arquivo 'bin/tp2.exe', dessa forma, o texto e a ordem lexicográfica contidos no arquivo "entrada.txt", que deve estar no diretório raiz, será executado. Além disso, o registro de memória do programa será armazenado no arquivo */tmp/textlog.out*.
- Caso o usuário prefira, é possível executar um arquivo específico de jogo de pôquer. Para isso, estando acessando o diretório raiz do programa, primeiramente é necessário executar o comando 'make build'. Em seguida, é preciso executar uma linha semelhante a seguinte, substituindo o nome do arquivo txt pelo arquivo desejado. Além disso, é preciso inserir o destino do arquivo de acesso de memória, após o '-p'.
    - 'bin/tp2.exe -i entrada.txt –o saída.txt -m 3 -s 2 -p /tmp/textlog.out'
- Em caso de dúvidas, é possível executar o comando 'bin/tp2.exe -h' para visualizar a mensagem de uso do programa.