Trabalho Prático 2
Ordenação em Memória Externa

Luís Felipe Ramos Ferreira
2019022553

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

lframos_ferreira@outlook.com

1. Introdução

A proposta do Trabalho Prático 2, da disciplina de Estruturas de Dados, foi implementar um algoritmo de ordenação que utilize tanto a memória principal (RAM) quanto a memória secundária(fita). Esse algoritmo é utilizado para ordenar uma lista de URLs, em forma decrescente de número de visitas à página.

Os algoritmos de ordenação utilizados foram quicksort e heapsort, como especificado pelo enunciado do trabalho. As entidades (URLs junto a seu número de visitas) são inicialmente lidas e armazenadas em fitas (arquivos de texto). Posteriormente, essas fitas são intercaladas, por meio do uso da estrutura de dados Heap, para gerar o arquivo final com as entidades corretamente ordenadas.

2. Implementação

O programa foi desenvolvido em um ambiente operacional Linux, na linguagem C++ compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estrutura de dados

A implementação do trabalho foi baseada em torno de uma lógica orientada a objetos, cujas ferramentas foram utilizadas para criação das classes Entidade e Heap, ambos sendo elementos importantes para a funcionalidade do código principal do trabalho.

Além disso, as funcionalidades principais do trabalho, como a leitura do arquivo de entrada, a escrita dos valores na memória secundária (fitas) e o intercalar das fitas (arquivos/rodadas) de entidades foram elaboradas em funções auxiliares, de forma a manter organizados os valores que forem lidos e criados durante a execução do programa.

2.2. Classes

Para a realização deste projeto, foram criadas duas classes, chamadas de Entidade e Heap, que representam elementos importantes para o funcionamento do código. A escolha por implementá-las em classes, artifício disponibilizado pela orientação à objetos da linguagem de programação C++, tem como fundamento a praticidade de implementação e maior organização das funcionalidades e atributos que estes elementos possuem.

A classe Entidade representa o tipo de dado entidade utilizado no trabalho. Sua representação é a de uma URL e o número de visitas que ela recebeu. Entretanto, outro atributo foi adicionado à ela para facilitar a implementação das funções do programa, sendo

ele o número que representa a fita (arquivo) de origem da entidade. Esse atributo torna o intercalamento das fitas mais prático e seguro contra falhas. Além de seus atributos, a classe possui métodos construtores e destrutores, além de métodos de acesso ao valor de seus atributos (*getters*). Por fim, na classe também são declaradas funções de tipo *friend*, que representam a sobrecarga dos operadores de comparação entre entidades, utilizados quando é desejado realizar a ordenação.

A classe Heap, por sua vez, é uma representação da estrutura de dados heap, especificada pelo enunciado como a estrutura que deve ser utilizada para intercalar as fitas. Ela tem como atributos o número de entidades guardados em si, assim como o vetor que armazena as entidades e representa a estrutura. Dentre suas funcionalidades, existem os seus métodos construtores e destrutores, assim como métodos de adição e retirada de itens. No entanto, destacam-se os métodos de criação do heap, composto pelos métodos *constroi()* e *refaz()*.

2.3. Ordenação

O trabalho tem como ponto principal a prática e a utilização de algoritmos de ordenação. Para seu funcionamento correto, utiliza-se o algoritmo quicksort para ordenar as entidades que serão armazenadas em cada uma das fitas (arquivos/rodadas) geradas após a leitura inicial do arquivo de entrada. A estrutura de dados Heap é então utilizada durante o intercalamento dessas fitas, com o objetivo de utilizar o método heapsort e, assim, conseguir que o arquivo de saída possua todas as entidades do arquivo de entrada ordenadas de maneira correta.

O algoritmo quicksort foi implementado utilizando as funções disponibilizadas pelos professores em aula, sendo elas *swap()*, *particao()*, *ordena()* e *quickSort()*. Assim como especificado, o algoritmo segue a lógica de dividir e conquistar, particionando o vetor de itens em vetores menores para facilitar a ordenação. O pivô, elemento do vetor em que será iniciado o particionamento, foi escolhido como o elemento do meio da estrutura, uma vez que essa decisão permite um melhor desempenho para o algoritmo na maioria dos casos.

Por sua vez, a estrutura Heap, assim como a anterior, foi implementada utilizando o código disponibilizado pelos professores, sendo este composto pelas funções *constroi()* e *refaz()*. Como supracitado, o heap foi implementado como uma classe e, portanto, suas funcionalidades foram implementadas dentro desta classe como métodos.

2.4. Programa

Para execução do programa como um todo, foram criadas quatro funções responsáveis por realizar os principais requisitos do trabalho. A função *leArquivoEntrada()* é chamada para ler o arquivo de entrada, que possui todas as entidades que devem ser ordenadas, e, conforme lê, utiliza a função *escreveFitaOrdenada()* para criar as fitas/rodadas, de forma já ordenada, que serão utilizadas no processo de intercalação.

A função *intercalaFitas()*, como o próŕio nome diz, é utilizada para as operações de intercalamento das fitas que possuem as entidades e, assim, escrever no arquivo de saída as entidades de forma ordenada. Essa função utiliza outra chamada *getFitas()*, que retorna um vetor armazenando todos os arquivos ifstream que representam as fitas abertas.

É importante salientar, em relação à execução geral do programa, que o número de fitas considerado na sua execução é sempre igual ao número de entidades contidas no

arquivo principal dividido pelo número máximo de entidades por fita. O atributo *numFitas*, lido pela linha de comando, não foi utilizado para análise de situações específicas em que esse valor é menor do que o necessário para o armazenamento de todas as entidades iniciais nas fitas.

3. Análise de Complexidade

3.1. Tempo

- **Método 'getURL' (classe Entidade):** esse método utiliza apenas operações constantes, portanto, possui complexidade de tempo O(1).

- **Método 'getNumVisitas' (classe Entidade):** esse método utiliza apenas operações constantes, portanto, possui complexidade de tempo O(1).

- **Método 'getNumArqOrigem' (classe Entidade):** esse método utiliza apenas operações constantes, portanto, possui complexidade de tempo O(1).

- **Método 'add' (classe Heap):** esse método utiliza o método 'constroi'. Portanto, sua complexidade de tempo é O(nlog(n)).

- **Método 'pop' (classe Heap):** esse método utiliza o método 'refaz', que possui complexidade logarítmica. Portanto, sua complexidade de tempo é O(log(n)).

- **Método 'constroi' (classe Heap):** esse método utiliza o método 'refaz', que possui complexidade logarítmica, para todos os nós internos do heap, isto é, para os n/2 primeiros elementos. Logo, sua complexidade de tempo é O(n log(n)).

- **Método 'refaz' (classe Heap):** esse método, no pior caso, percorre todo um galho da árvore binária. Logo, sua complexidade de tempo é O(log(n)).

- **Método 'vazio' (classe Heap):** esse método utiliza apenas operações constantes, portanto, possui complexidade de tempo O(1).

- **Método 'getNumElementos' (classe Heap):** esse método utiliza apenas operações constantes, portanto, possui complexidade de tempo O(1).

- **Função 'swap' (QuickSort):** essa função utiliza apenas operações constantes, portanto, possui complexidade de tempo O(1).

- **Função 'quickSort' (QuickSort):** essa função utiliza a função ordena para todo o vetor que deve ser ordenado. Ela possui, em caso médio, uma complexidade de tempo O(n log(n)).

- **Função 'leArquivoEntrada' :** essa função percorre um laço de tamanho igual ao número n de entidades no arquivo de entrada. Toda vez que o contador chega ao parâmetro m (número de entidades por fita) e quando o arquivo de entrada acaba, as funções quicksort() e escreveFitaOrdenada() são chamadas.

Dessa maneira, a complexidade de tempo desta função é dada por n + (n/m) * m * log(m) = n + n * log(m). Portanto, a complexidade de tempo da função é O(n*log(m)).

- **Função 'escreveFitaOrdenada' :** essa função percorre um laço de tamanho 'n', portanto, sua complexidade de tempo é linear, isto é, O(n).

- **Função 'intercalaFitas' :** essa função chama a função getFitas(), que possui complexidade linear, e, após isso, percorre um laço de tamanho n. Então, é chamado o método constroi() para o heap, que possui complexidade O(mlog(m)), onde 'm' é, neste caso, o número de elementos no heap. Por fim, um laço de tamanho igual ao número total de entidades 't' no arquivo de entrada é percorrido e, para cada execução do laço, é chamado o método pop() para o heap, de complexidade O(log(m)), onde 'm' é o número de entidades no heap, e o método add(), de complexidade O(mlog(m)). Portanto, temos que a função de complexidade dessa função é n + n + mlog(m) + tlog(m) + tmlog(m), ou seja, pelas regras, sua complexidade de tempo é O(n + (t + m + tm)log(m)).

- **Função 'getFitas' :** essa função percorre um laço de tamanho n, portanto, sua complexidade de tempo é linear, isto é, O(n).

3.2. Espaço

- **Método 'getURL' (classe Entidade):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'getNumVisitas' (classe Entidade):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'getNumArqOrigem' (classe Entidade):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'add' (classe Heap):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'pop' (classe Heap):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'constroi' (classe Heap):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'refaz' (classe Heap):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é

O(1).

- **Método ‘vazio’ (classe Heap):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘getNumElementos’ (classe Heap):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘swap’ (QuickSort):** essa função realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘quickSort’ (QuickSort):** essa função realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘leArquivoEntrada’:** essa função cria um vetor auxiliar de tamanho n, o qual armazenará as entidades que devem ser ordenadas. Portanto, sua complexidade de espaço é linear, isto é, O(n).

- **Função ‘escreveFitaOrdenada’ :** essa função realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘intercalaFitas’ :** essa função cria um heap que, por sua vez, possui um vetor de tamanho n em sua composição. Portanto, sua complexidade de espaço é O(n).

- **Função ‘getFitas’ :** essa função cria um vetor auxiliar de tamanho n, o qual armazenará as entidades que devem ser ordenadas. Portanto, sua complexidade de espaço é linear, isto é, O(n).

4. Análise experimental

A análise de desempenho do código foi feita utilizando a biblioteca ‘memlog.h’ disponibilizada pelos professores. Com o intuito de obter resultados em um formato desejado para a análise deste trabalho especificamente, foram feitas algumas alterações nos valores que são escritos no arquivo de registro de acesso.

Para gerar os gráficos e tabelas dos resultados obtidos, o que permite uma melhor análise dos dados e informações, foi utilizada a ferramenta de linha de comando *gnuplot*. Ela foi escolhida pela facilidade de gerar e manipular os gráficos, utilizando comandos simples e rápidos. A documentação deste recurso está disponibilizada nas referências bibliográficas do trabalho.

Além disso, para a geração de arquivos de entrada com tamanhos variados, que pudessem ser utilizados para a análise de performance, foi utilizada a biblioteca *geracarga*, disponibilizada pelos professores.

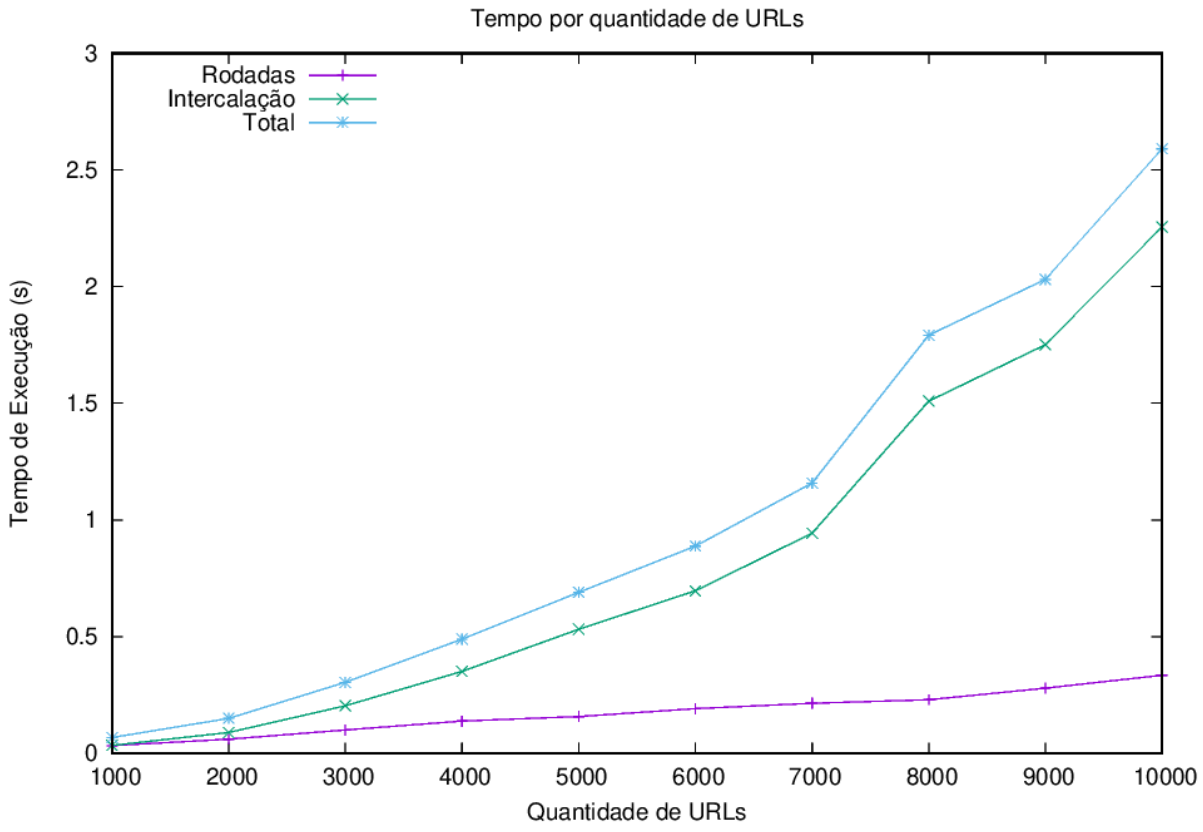
4.1 Desempenho computacional

Em relação ao desempenho computacional, foram marcados os tempos de execução da geração de rodadas, da intercalação das fitas geradas e do total do programa. Para isso, utilizou-se as funções disponibilizadas na biblioteca ‘memlog.h’.

Com o intuito de gerar um gráfico preciso e correto com os valores obtidos, foram feitos testes com arquivos de entrada do programa de 10 tamanhos diferentes. O menor arquivo testado possuía 1000 (mil) entidades, enquanto cada arquivo sequente possuía 1000 (mil) entidades a mais que o anterior. Para cada um destes arquivos, foram feitos 10 testes, e a média de tempo de execução destes testes foi utilizada para montagem do gráfico, com o fito de obter maior precisão nos valores.

Em primeiro lugar, consideramos o caso em que o número de fitas é sempre maior ou igual à função teto da divisão entre o número de entidades no arquivos de entrada e o número de entidades máximo permitido por fita.

Partindo desse pressuposto, utilizamos o algoritmo de modo que o número máximo de entidades por fita seja 3, e, dessa maneira, o seguinte gráfico de tempo de execução foi obtido:

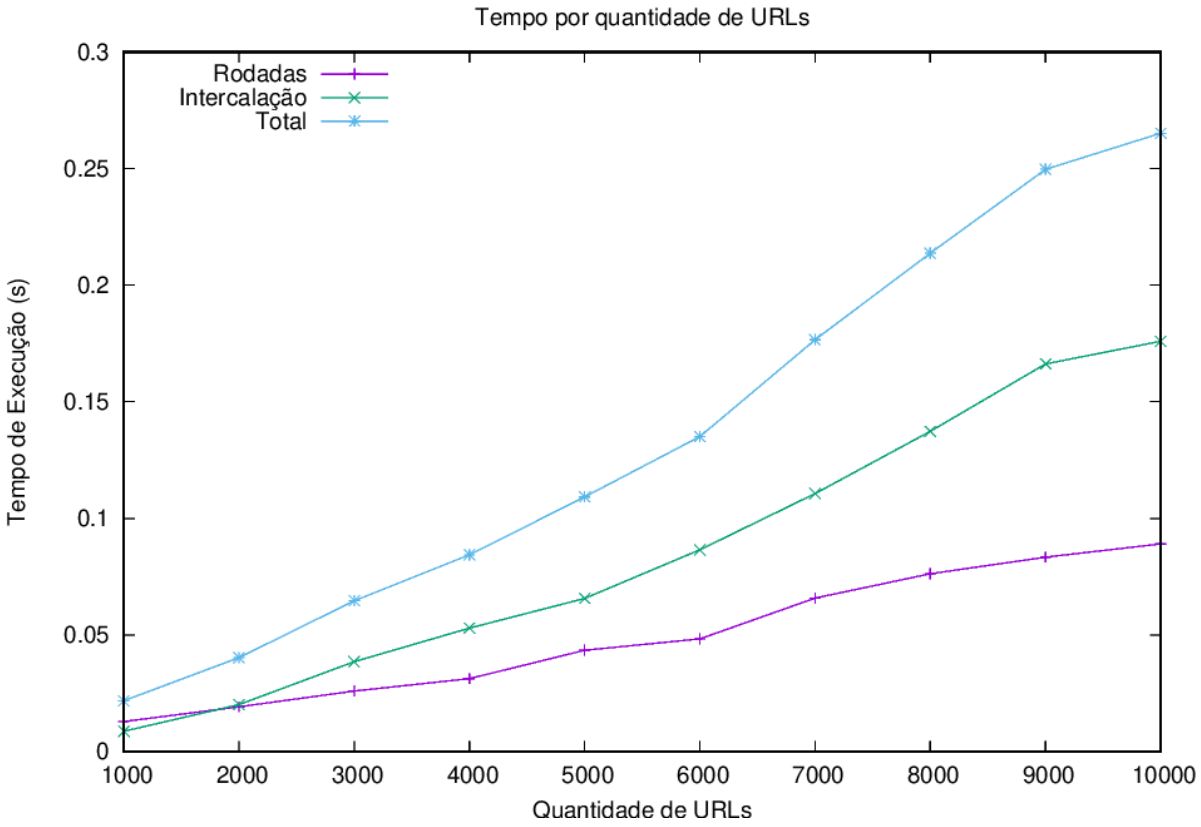


Podemos notar que os resultados obtidos foram satisfatórios. O gráfico segue a lógica prevista para o funcionamento do código a partir da análise da complexidade de tempo

estudada. Isto é, a geração de rodadas é muito mais rápida que a intercalação de rodadas, uma vez que a complexidade de tempo da intercalação possui um componente quadrático multiplicado por um logarítmico, enquanto a geração de rodadas possui um componente linear multiplicado por um logarítmico. Nota-se, portanto, que com o aumento do número de entidades no arquivo principal, o tempo cresce junto, seguindo um padrão gráfico observado pela análise anteriormente.

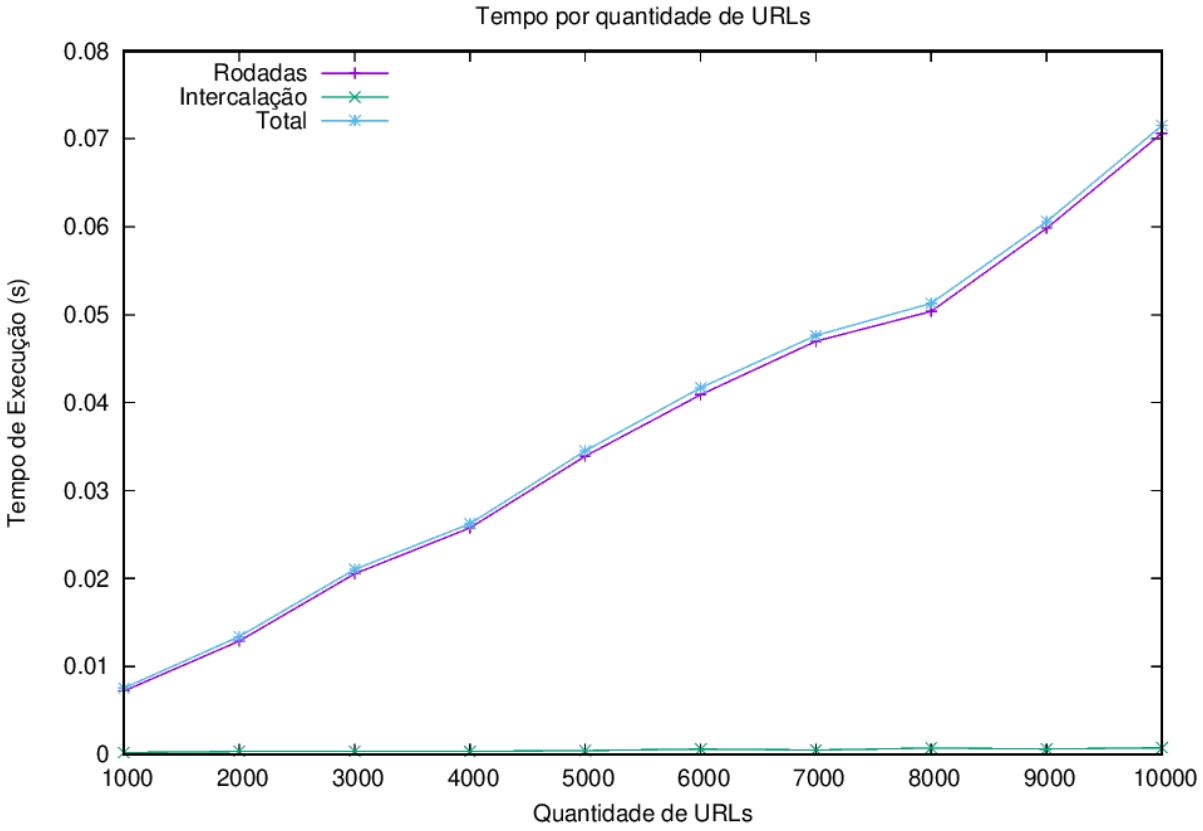
Destaca-se, então, a diferença de tempo entre a intercalação das fitas e a geração das rodadas. Vale notar que o número máximo de entidades por arquivo afeta diretamente o tempo de execução do programa.

Pode-se confirmar isso gerando um gráfico com os mesmos valores, mas, dessa vez, o número máximo de entidades por fita será 100 (cem). Após os testes, temos o seguinte gráfico como resultado:



Podemos ver, por meio do gráfico acima, que com o aumento do número máximo de entidades permitido por fita, o algoritmo se torna mais rápido e eficiente, uma vez que são necessárias menos aberturas de arquivos para escrita e leitura, assim como a intercalação se torna muito mais rápida. Portanto, conclui-se que o número de entidades por fita, que irá ditar quantas fitas são criadas, é um dos mais importantes fatores no que se refere à eficiência do programa.

Agora, vamos considerar um caso base onde o número de entidades máximo por fita é maior que o número total de entidades no arquivo de entrada. Nesse cenário, apenas uma fita é criada e não é necessário utilizar a intercalação. Vamos utilizar como base as mesmas quantidades de entidades iniciais utilizadas nos gráficos anteriores.



Com o gráfico gerado, fica evidente que, neste caso, o que dita o tempo total de execução do programa é a geração da rodada única que será utilizada,ou seja, o tempo depende basicamente da leitura do arquivo de entrada e o uso do algoritmo de ordenação *quicksort*. Vemos que o tempo de execução da intercalação é praticamente nulo, já que a intercalação de fitas em si não é executada, evitando o aumento drástico do tempo de execução.

5. Estratégias de robustez

Para tratar falhas e deixar o código mais robusto, foi utilizada a biblioteca ‘msgassert.hpp’, disponibilizada pelos professores. Essa biblioteca define duas ‘macros’ para tratar os possíveis erros encontrados no uso das funções do programa.

Ela foi utilizada em funções e métodos onde foi julgado necessário garantir que parâmetros de entrada fossem dados da forma esperada. Ela também foi utilizada para garantir que os arquivos utilizados na implementação fossem abertos corretamente.

6. Conclusão

Este trabalho prático teve como propósito criar um programa que ordene em forma decrescente entidades lidas de um arquivo de entrada, utilizando tanto a memória principal como a memória secundária. No caso deste trabalho, as entidades são compostas por uma URL e seu número de visitas. A linguagem de programação escolhida para a implementação foi C + +, uma vez que permite o uso de um paradigma orientado a objetos, além de ser uma linguagem com inúmeras outras funcionalidades.

As entidades foram implementadas por meio de uma classe, assim como a estrutura de dados *heap*, que é utilizada durante a última fase do programa, para a intercalação das fitas da memória secundária. O algoritmo de ordenação utilizado para geração das fitas foi o Quicksort, como requer o enunciado. O pivô escolhido para seu uso foi o elemento central do vetor, para garantir maior eficiência em casos gerais.

Pode-se perceber que o trabalho aborda questões importantes para o desenvolvimento de sistemas, como algoritmos de ordenação e o uso de memória. Tendo isso como consideração, nota-se que as estratégias para resolução do problema proposto foram pensadas com o intuito de tornar seu funcionamento o mais eficiente e correto possível.

Dessa forma, é importante ressaltar que o desenvolvimento deste código foi importante para o estudo e aprimoramento de diversos conceitos ligados à computação. Dentre eles, se destaca a forma como funciona a memória de um computador, isto é, como e onde os dados e informações são armazenados e como acessá-los, além do uso de algoritmos de ordenação, funcionalidades utilizadas em qualquer sistema e extremamente importantes. Além disso, frisa-se o desafio de se utilizar e manipular arquivos durante a resolução do problema proposto, o que permitiu um melhor entendimento de como esses dados funcionam. Por fim, é necessário salientar o uso da estrutura *Heap* que, mesmo com códigos já disponibilizados pelos professores, exigiu muita refatoração e revisão, para que enfim funcionasse como desejado.

7. Referências bibliográficas

GEEKSFORGEEKS. Core Dump (Segmentation fault) in C/C++.geeksforgeeks. Disponível em: <https://www.geeksforgeeks.org/core-dump-segmentation-fault-c-cpp/>. Acesso em: 12/01/2022.

GEEKSFORGEEKS. Heap Data Structure.geeksforgeeks. Disponível em: <https://www.geeksforgeeks.org/heap-data-structure/>. Acesso em: 24/11/2021.

MIZRAHI, Victorine Viviane. Treinamento em linguagem C. 2ª Edição. Editora Pearson.

DROZDEK, Adam. Estrutura de dados e algoritmos em c++. 4ªEdição. Editora Cengage Learning.

IME.USP. Paulo Feofiloff. Ordenação: Quicksort. Disponível em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/quick.html>. Acesso em 16/01/2022.

UFSC. Operações com Arquivos. Disponível em: <https://moodle.ufsc.br/pluginfile.php/2377820/mod_resource/content/0/exercisios%20arquivos.pdf>. Acesso em 11/01/2022.

GEEKSFORGEEKS.Operator Overloading in C/C++.geeksforgeeks. Disponível em: <https://www.geeksforgeeks.org/operator-overloading-c/>. Acesso em: 09/01/2022.

UNESP. Tutorial: introdução ao uso do aplicativo Gnuplot.UNesp. Disponível em: <http://www2.fct.unesp.br/docentes/carto/galo/web/gnuplot/pdf/2017_Galo_Gnuplot_Tutori

al.pdf>. Acesso em: 18/01/2022.

GNUPLOT.Gnuplot homepage. gnuplot. Disponível em: <http://www.gnuplot.info/>. Acesso em: 19/01/2022.

8. Compilação e execução

- Acesse o diretório TP
- No terminal, utilize comando *make* para compilar os arquivos e gerar o 'target' do programa.
- Para utilizar o programa, digite no terminal ‘bin/programa’ seguido do nome do arquivo de texto de entrada, o nome do arquivo de texto de saída e o número de entidades que deve ser aceito em cada fita. Também é possível digitar mais um valor como próximo argumento, o qual será armazenado como número de fitas, mas não será utilizado pelo programa.
- **Exemplo:** bin/programa entrada.txt saida.txt 50
- Para apagar os arquivos ‘object’ e o ‘target’, assim como as fitas armazenadas na pasta ‘rodadas’, utilize no terminal o comando make clean.