Trabalho Prático 2
Ordenação em Memória Externa

Vinicius Trindade Dias Abel
Matrícula: 2020007112

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

viniciustda@ufmg.br

1. Introdução

O problema proposto foi implementar um algoritmo de ordenação que utilize não só a memória principal, mas também a memória secundária para que um computador pessoal seja capaz de processar um grande volume de dados sem exceder a memória principal.

Desta forma, o algoritmo recebe como entrada um arquivo contendo URLs e o número de visitas que cada uma recebe (este conjunto é chamado de entidade) e seu funcionamento é dividido em duas partes. Na primeira, ele lê um determinado número de entidades que é ordenado na memória principal e, em seguida, armazenado em um arquivo rodada-n.txt chamado fita. Na segunda, as n fitas são intercaladas, gerando um arquivo final com todas as entidades ordenadas de forma decrescente pelo número de visitas que cada URL recebeu.

Por fim, o algoritmo recebe como parâmetro o nome do arquivo que será ordenado, o nome do arquivo final, o número de entidades que serão armazenados em cada fita e, opcionalmente, o número máximo de fitas que podem ser geradas. Assim, é possível trabalharmos com os parâmetros numero de entidades e numero de fitas, para definir alguns casos:

1. Número de entidades > 0 e Número de fitas = 0 ou Número de fitas não é passado: todas as entidades são lidas e são geradas quantas fitas forem necessárias.
2. Número de entidades = 0 e Número de fitas > 0: Todas as entidades são lidas e distribuídas igualmente entre as n fitas que podem ser geradas.
3. Número de entidades > 0 e Número de fitas > 0: Deste modo podemos definir quantas entidades serão lidas, porque o algoritmo para de ler as entidades ao atingir o número máximo de fitas permitidas.

2. Método

2.1. Estrutura de Dados

A implementação do programa teve como base da estrutura de dados os algoritmos de ordenação QuickSort e HeapSort que são aplicados em vetores da classe Entidade.

Para esta implementação foram criadas classes (detalhadas no próximo tópico), sendo elas: Arquivo, Fila e FilaEncadeada, Entidade. Cada classe foi criada em um arquivo .hpp (e implementadas em arquivo .cpp de mesmo nome) separado das outras.

Além destes, foram criados um arquivo .hpp e um .cpp de nome arq_funcao que possui funções que trabalham com arquivos, um arquivo .hpp e um .cpp de nome quicksort que possui o algoritmo quicksort e uma função que coloca em ordem alfabética as entidades que possuem o mesmo número de visitas, um arquivo .hpp e um .cpp de nome heap que possui o algoritmo heapsort, um arquivo .hpp e um .cpp de nome primeira_etapa que realiza a primeira etapa do tp (cria as fitas já preenchidas e ordenadas), e um arquivo .hpp e um .cpp de nome segunda_etapa que realiza a segunda etapa do tp (intercala as fitas no arquivo de saída).

2.2. Classes

Arquivo

A classe Arquivo possui uma variável conteudo do tipo string, que é onde fica armazenado uma linha do arquivo que está sendo trabalhado.

Também possui variável prox do tipo Arquivo que é um ponteiro que aponta para o próximo da fila.

Além de disso, a classe possui apenas as funções construtor e destrutor.

Fila e FilaEncadeada

Estas classes, onde FilaEncadeada é uma classe filha de Fila, foram vistas em aula. Entretanto sofreram algumas alterações para se adequarem a solução do problema.

A primeira alteração é que todos os elementos da lista são do tipo Arquivo.

Outra alteração foi que apenas as funções Vazia, GetTamanho, Enfileira, Desenfileira e limpa foram implementadas (além do construtor e destrutor).

Entidade

A classe Entidade possui uma variável conteudo do tipo string, que é onde fica armazenado a entidade (URL + número de visitas).

Também possui uma variável visitas do tipo int, que armazena apenas o número de visitas da entidade. É usada para a ordenação.

E possui uma última variável rodada do tipo int, para armazenar em qual fita a entidade está inserida. É usada na segunda parte do tp, quando insere a entidade no heap.

2.3. Main

A função main lê os parâmetros passados pela linha de comando, verifica se foi limitada a quantidade de fitas que podem ser criadas, conta quantas entidades existem no arquivo de entrada e por fim, chama as funções PrimeiraEtapa e SegundaEtapa, que são, como o nome diz, a primeira e segunda etapa do tp respectivamente.

2.4. Configuração para teste

– Sistema Operacional do seu computador: Windows 11;
– Linguagem de programação implementada: C++;
– Compilador utilizado: G++ da GNU Compiler Collection;
– Dados do seu processador: i7;
– Quantidade de memória primaria: 16GB;
– Quantidade de memória secundaria: 512GB.

3. Análise de Complexidade

função ArquivoVazio - complexidade de tempo: esta função possui um while que é executado uma ou duas vezes apenas, independendo do tamanho do arquivo. Este while apenas lê uma linha do arquivo e incrementa o contador, que ao passar de 1 dá um break no while. Dessa forma, a complexidade assintótica de tempo desta função é Θ(1).

função ArquivoVazio - complexidade de espaço: no pior caso esta função executa o while duas vezes e usa armazena apenas uma entidade. Assim, a complexidade assintótica de espaço desta função é Θ(1).

função ApagaPrimeiraLinhaArquivo - complexidade de tempo: esta função possui dois laços separados de tamanho n. Cada um destes laços tem complexidade O(n). Dessa forma, a complexidade assintótica de tempo da função é Θ(n), já que um laço não está dentro do outro.

função ApagaPrimeiraLinhaArquivo - complexidade de espaço: esta função realiza operações com uma fila, que possui tamanho n-1. Assim, a complexidade assintótica de espaço desta função é Θ(n-1).

função ContaLinhas - complexidade de tempo: esta função possui um laço de tamanho n, sendo n o número de linhas do arquivo. Logo a complexidade assintótica de tempo desta função é Θ(n)

função ContaLinhas - complexidade de espaço: esta função possui um laço de tamanho n, sendo n o número de linhas do arquivo. E dentro do while

existe um contador que é incrementado a cada execução do laço. Logo a complexidade assintótica de espaço desta função é Θ(n)**.**

função QuickSort - complexidade de tempo: como visto em aula, tem em média uma complexidade O(n log n) e por usar a mediana de três, evita o pior caso.

função QuickSort - complexidade de espaço: complexidade seria O(n) por trabalhar com um vetor de tamanho n.

função PrimeiraEtapa - complexidade de tempo: esta função possui um while que é executado n vezes. Dentro do while existem laços, mas nenhum laço possui outro laço dentro, e também é chamado o algoritmo de ordenação QuickSort. Portanto, a complexidade é $\Theta(n^2)$.

função PrimeiraEtapa - complexidade de espaço: esta função trabalha com um vetor de tamanho n. Então a complexidade é a complexidade é Θ(n).

função HeapSort - complexidade de tempo: como visto em aula, tem uma complexidade O(n log n) sempre.

função HeapSort - complexidade de espaço: complexidade seria O(n) por trabalhar com um vetor de tamanho n.

função SegundaEtapa - complexidade de tempo: esta função possui dois whiles separados de tamanho n. em cada while as funções HeapSort e ApagaPrimeiraLinhaArquivo são chamadas separadamente. Desta forma a complexidade da função SegundaEtapa é $\Theta(n^2)$.

função SegundaEtapa - complexidade de espaço: esta função trabalha com um vetor de tamanho n. Então a complexidade é a complexidade é Θ(n).

Geral ou função main - complexidade de tempo: A função main não possui laços, e chama uma vez as funções ContaLinhas, PrimeiraEtapa e SegundaEtapa. Como a maior delas possui complexidade $\Theta(n^2)$, a complexidade geral também é $\Theta(n^2)$.

Geral ou função main - complexidade de espaço: A função main não possui laços, e chama uma vez as funções ContaLinhas, PrimeiraEtapa e SegundaEtapa. Como a maior delas possui complexidade Θ(n), a complexidade geral também é Θ(n).

4. Estratégias de Robustez

As classes foram implementadas de forma que apenas as funções são públicas, para evitar o acesso indevido aos atributos.

Se ocorrer algum erro na abertura de arquivo, o erro é indicado.

As etapas do tp não foram implementadas diretamente no main para evitar que o código de cada etapa seja alterado e consequentemente não cumpra sua função. Além de facilitar o entendimento do código e deixar bem definida cada etapa.

Devido as dúvidas em relação ao parâmetro de entrada que define o número de fitas, ele pode ou não ser passado. Sendo que a sua forma de uso no código foi descrita na introdução.

Foram criadas diversas classes e bibliotecas, que facilitam o entendimento do código e permite que sejam usadas em futuros trabalhos. Como foi o caso da FilaEncadeada, usada no tp1.

O tp aceita arquivos de entrada que terminam com uma linha vazia ou que terminam com uma entidade.

5. Análise Experimental

Desempenho computacional: Está ligado ao número de entidades que existem no arquivo de entrada, quanto mais entidades, maior o tempo de execução.

Acesso à memória: O programa foi implementado de forma que o tamanho do vetor é definido pelo número de fitas criadas. Dependendo do tamanho do arquivo de entrada e do parâmetro passado, a memória utilizada altera. Quanto mais fitas forem criadas, mais memória será usada. Contudo vale ressaltar que o número de fitas é muito menor que o número de entidades, o que cumpre a função do programa que é não deixar exceder a memória principal. (Caso seja definido apenas uma fita para todas as entidades, esta observação deixa de ser verdadeira, pois o vetor será de tamanho n, sendo n o número de entidades)

6. Conclusões

Este trabalho lidou com um algoritmo de ordenação que utiliza não só a memória principal, mas também a memória secundária para processar um grande volume de dados sem exceder a memória principal. Na qual a abordagem utilizada para sua resolução foi a utilização de um vetor, dos algoritmos de ordenação QuickSort e HeapSort, e as fitas.

Com a solução adotada, pode-se verificar que o programa busca aceitar e realizar sua função com diferentes parâmetros na entrada, se adequando as possibilidades.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados a fila, manipulação de strings e de arquivos passados como parâmetro, a criação de estruturas de dados, classes e funções, e a trabalhar com algoritmos de ordenação.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo a manipulação do arquivo .txt, trabalhar com os parâmetros passados por linha de comando, entender o funcionamento dos algoritmos de ordenação à ponto de conseguir modificá-los para atender a necessidade do problema, criar classes e funções para resolver problemas que apareciam, e a implementação do makefile.

7. Bibliografia

- std:basic_string. Disponível em: https://www.inf.pucrs.br/~flash/lapro2ec/cppreference/w/cpp/string/basic_string.html. Acesso em: 14 de dezembro de 2021.

- C++ - Operações com arquivos. Disponível em: https://moodle.ufsc.br/pluginfile.php/2377820/mod_resource/content/0/exercicios%20arquivos.pdf. Acesso em: 14 de dezembro de 2021.

- Slides do professor.

8. Instruções para compilação e execução

• Acesse o diretório TP

• Utilizando o terminal, compile o TP2 utilizando o comando: make

• Com esse comando, será gerado o arquivo tp2.exe no diretório bin e os arquivos *.o no diretório obj

• Acesse o diretório bin

• Utilizando o terminal, execute o arquivo tp2.exe junto com o arquivo de entrada *.txt e o arquivo de saída *.txt, inserindo também o número de entidades que serão armazenadas em cada fita e, se quiser, insira o número máximo de fitas utilizando o comando: tp2.exe <arq_entrada.txt> <arq_saida.txt> <num_entidades_fita> <num_max_fitas(opicional)>