Trabalho Prático 2
Ordenação em memória externa

1. Introdução

O problema proposto foi implementar um programa, que utiliza-se a memória externa na ordenação de páginas da web de acordo com o sua visualização.

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

2.1 Configurações do Sistema de Desenvolvimento

Sistema Operacional: Linux - PopOS!
Linguagem de programação: C++
Compilador: g++ versão 11.2.0
Hardware: Processador Ryzen 5-5500U, 8GB de memória RAM

2.1 Estruturas de dados

As estruturas de dadas utilizas para a resolução deste trabalho foram lista estática, pilha e a classe heap.

As lista estáticas foram usadas frequentemente, devido as características do projeto que em diversas partes do programa é necessário um número fixo de variáveis. Ela foi utilizada para a armazenagem da entrada, para gerenciar os arquivos de entrada e saída e como estrutura principal da classe heap.

A pilha foi utilizada na implementação do algoritmo quicksort iterativo, e possui os métodos empilha(), desempilha(), e vazia().

Como dito anteriormente foi utilizado uma lista estática para a implementação da classe heap, esta classe possui o diferencial que a cada nova inserção na lista, ela é ordenada e o método remove() remove o ultimo elemento da lista, ou seja o maior elemento.

O arquivo structs.cpp, contém as structs foram necessárias para a solução deste trabalho. Como por exemplo a struct que salva a url e o numero de acessos.

2.2 Algoritmos de Ordenação

O algoritmo de ordenação utilizado no trabalho foi o quicksort, ele foi utilizado para ordenar os dados enquanto eles estivem na memoria interna. Para a sua implementação iterativa, foi necessário utilizar a pilha como dito anteriormente. A escolha do pivo se deu atravez da mediana de três valores o inicio, meio e o fim, afim de se evitar seu pior caso. Para subvetores menores que 4 elementos é utilazado o método de ordenação inserção para uma maior efiencia.

A ordenação por inserção também foi utilizado neste trabalho, em conjunto do quicksort, pois para entradas pequenas o método da inserção é mais eficiente.

3. Análise de complexidade

Nesta sessão foi realizada a análise das principais funções e métodos que contribuíram para as funções chaves do programa.

Funções e métodos frequentemente utilizados da biblioteca padrão, e operações relacionadas com o uso de string serão considerados como tendo custo constante e operações.

3.1 Heap.hpp

insere(): Insere um novo elemento no vetor e o ordena.
Complexidade de tempo:
O método é O(n log n), pois o custo de se inserir um novo elemento é O(1) mais o custo de se ordenar pelo quicksort é O(n log n)

remove(): Remove o último elemento do vetor

Complexidade de tempo:

O método possui complexidade O(1), pois ele acessa o elemento por um vetor estático O(1), decrementa em um o tamanho e o índice do ultimo item.

Complexidade de espaço:

O método é O(1), pois necessita criar somente uma nova variável que recebe o elemento retirado, as demais variáveis são atributos da classe

vazio(): Verifica se o heap está vazio

Complexidade de Tempo:

O método é O(1), pois realiza uma comparação, comparando se o atributo tamanho é igual a 0;

Complexidade de espaço:

O método é Θ(1), pois não é utilizada nenhuma memória extra, é feita somente uma comparação.

3.2 quicksort.hpp

Complexidade de Tempo:

O método é O(n log n), pois como foi discutido em aula o custo do pior caso do quicksort é $O(n^2)$, porem como o pivô é a mediana de três valores a probabilidade de se cair no pior caso é ínfima, e portanto usaremos o custo do caso médio O(n log n).

Complexidade de Espaço:

O método é O(1), pois são criados um numero fixo de variáveis.

3.3 main.cpp

imprimi_arr():

Complexidade de Tempo:

A função é O(n), pois itera sobre o vetor “arr” n vezes, a cada iteração o conteúdo de um elemento do vetor é imprimi na saída tendo custo O(1).

Complexidade de Espaço:

A função é O(1), pois apesar de não ter sido criado nenhuma nova variável na função, foi criado um variável do tipo ponteiro para a armazenagem do endereço do primeiro elemento do vetor.

le_entrada():

Complexidade de Tempo:

A função é O(n), pois no pior caso itera n vezes, e cada iteração le a entrada e o salva em um vetor tendo custo O(1).

Complexidade de Tempo:

A função é O(1), pois é criado um numero fixo de variáveis e o vetor utilizado já foi criado anteriormente

get_input(): Separa a entrada obtida pelo getline

Complexidade de Tempo:

A função é O(1), pois ocorrem somente operações com string.

Complexidade de Espaço:

A função é O(1), pois cria um numero fixo de variáveis que não depende da entrada

gera_rodadas(): Separa a entrada em arquivos externos

Complexidade de Tempo:

A função é O(m log n), pois o laço é executado n_entrada(numero de entrada contidas no arquivo) / n_entidades (numero de entidades a serem lidas por arquivo) vezes, e cada iteração são executadas

funções com custo O(n) e O(n log n), com n sendo n_entidades. Como a quantidade de vezes que o laço é executado está relacionado com o tamanho n, a complexidade da função gera_rodadas ficará

[m (n_entrada) / n] * max(O(n), O(n log n) = m * O(n log n) / n => O(m log n).

Complexidade de Espaço:

A função é O(n), pois é criado um numero fixo de variáveis que não depende da entrada, mais um vetor de inteiros dependente da entrada.

intercala(): Ordena “n” fitas, em um único arquivo

Complexidade de Tempo:

A função é O(mn), pois são executados 4 laços, 3 sendo laços for e 1 sendo laço while. Os laços for iteram n (com n sendo um parâmetro da função) vezes sobre blocos de comando com custo constante O(1) e o laço while executa m * n vezes (com m sendo a quantidade de dados presentes em um arquivo) um conjunto de comandos com as funções de custo heap::remove O(1), get_input O(1).

$$3*O(n) + O(mn) = O(mn)$$

Complexidade de Espaço:

A função é O(n), pois o numero de variáveis criadas dentro do escopo da função dependem de um parâmetro passado para a função.

intercala_rodadas_maior_fitas():

Complexidade de Tempo:

A função é O(mnp), pois os comando fora dos laços são O(np), do laço externo sem contar com o laço interno é O(1) e o laço interno é O(np). Os laços internos e externos variam em função de numero de rodadas (m) e o numero de fitas (n). No pior caso n será igual a dois, e portanto será necessário um numero maior de intercalações até o resultado final. O laço externo é executado $\log_2(m)$ e o interno é executado $m / 2^x$ a cada iteração, com x = 1 e a cada iteração ele aumenta em um. Portanto o laço interno é executado $\sum_{i=1}^{\log_2(m)} \frac{m}{2^i}$, substituindo $\log_2(m)$ por k, e fazendo $k \rightarrow \infty$, teremos uma série geométrica cujo o resultado é m, logo o laço externo é interno p vezes, com p <= m. O bloco de comando de laço interno é O(np), resultando em m * O(np) = O(mnp).

$$O(mnp) + O(np) = O(mnp)$$

Complexidade de Espaço:

A função é O(1), pois as variáveis criadas não dependem de nenhum parâmetro.

main():

Complexidade de Tempo:

A função é O(m log n) + O(npq), pois a função gera_rodadas é O(m log n) (m = numero de dados do arquivo, n = numero de entidades a serem lidos) e no pior quando rodadas forem maior que o numero de fitas a função a ser executada será intercala_rodadas_maior_fitas tendo custo O(npq) (p = numero de rodadas, q = numero de fitas, n = numero de entidades).

Complexidade de Espaço:

A função é O(1), pois nenhuma variável criada depende de algum parâmetro.

4. Estratégia de Robustez

Foi utilizado neste projeto a biblioteca msgassert para verificar inconsistências e pré-condições.

Somente as principais funções

Arquivo main.cpp:

Foi utilizado para verificar se o arquivo de entrada foi aberto corretamente, caso contrário o programa é interrompido e uma mensagem é mostrada no terminal

Função parse_args:

Verifica se os parâmetro passados para o programa estão corretos, os parâmetro numéricos são mesmo números, se então dentro de um intervalo adequado, caso não o programa é interrompido e uma mensagem é mostrada no terminal

Função gera_rodadas:

Verifica se o parâmetro que decide o numero de entidades por arquivo é um numero positivo, caso não o programa é interrompido e uma mensagem é mostrada no terminal.
Essa pré-condição deve ser sempre atendida, pois ela já foi atendida no parse_args().
Também verifica se os arquivos de saída “rodadas” foram abertos corretamente através da função escreve_arquivo(), caso não o programa é interrompido e uma mensagem é mostrada no terminal.

Função intercala:
Verifica se os arquivos foram abertos, caso não o programa é interrompido e uma mensagem é mostrada no terminal.

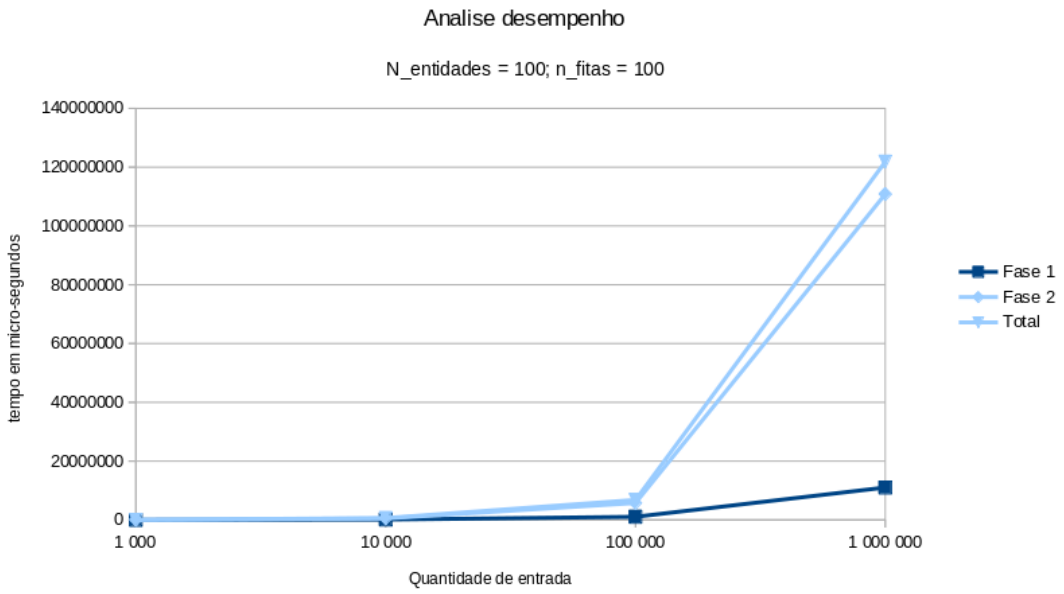
Função intercala_rodadas_maior_fitas():
Verifica se o numero de fitas e o numero de rodas são numero válidos, numero de fitas deve ser pelo menos dois para que posso ocorrer a intercalação, caso contrário o programa é interrompido e uma mensagem é mostrada no terminal.

5.Análise Experimental

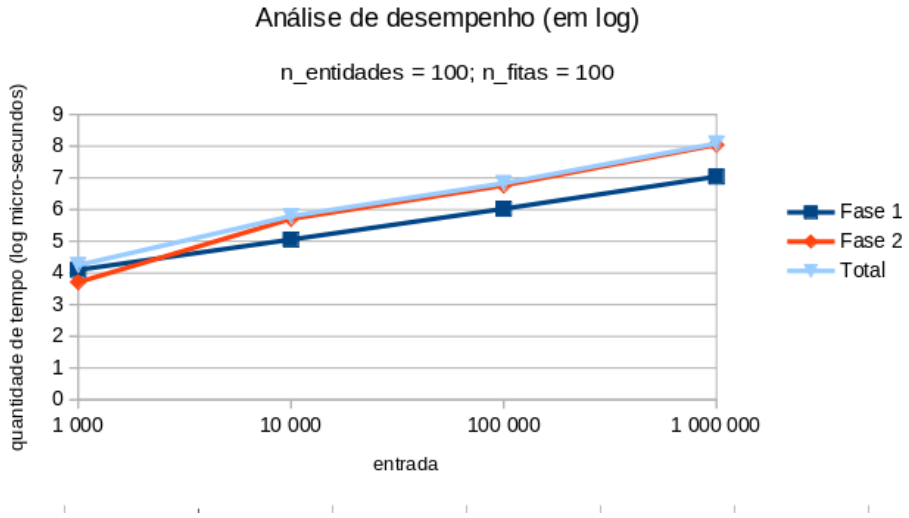
Desempenho real do código em relação ao tempo.

	1 000	10 000	100 000	1 000 000
Fase 1	12513	113486	1059003	11025473
Fase 2	5087	509128	5742400	110829719
Total	17600	622614	6801403	121855192

Utilizando arquivos de tamanho de vários tamanhos, percebemos que na fase 1 (gera rodadas) que possui a complexidade de O(m log n), cresce em uma velocidade menor que a fase 2 (intercala) de complexidade O(mnp).

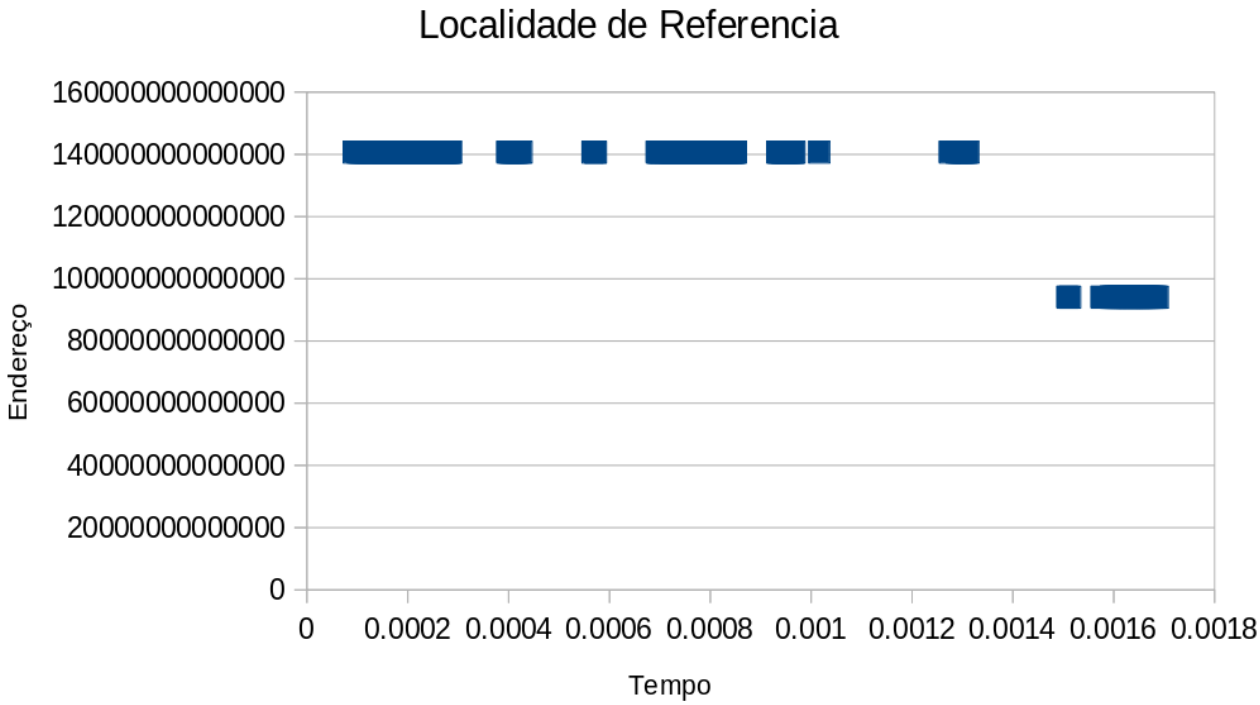


Refazendo o gráfico usando o eixo tempo como log(tempo), podemos perceber melhor esse fato.



Desempenho real do gráfico em relação ao espaço.

Não foi possível utilizar a biblioteca memlog em conjunto com o programa analisa, para que pudesse obter uma melhor análise, portanto foi para a geração do gráfico foi utilizado as informações de tempo e de endereço, contidos no arquivo gerado pelo memlog.
Pelo gráfico conseguimos as informações de quais blocos de memória estavam sendo utilizados em determinado momento, além de ficar bastante visível a fase 1 e 2 do programa. Com a fase 1 utilizando os endereços que inicializam com 14 e a fase 2 os endereços que inicializam com 9.



6. Conclusão

Este programa lidou com a manipulação e organização de dados no memória externa.

Os maiores desafios na implementação deste trabalho foram as intercalação intermediárias, quando a quantidade de fitas é menor que o número de rodadas e a implementação do quicksort iterativo apesar já ter sido mostrado em aula o código do algoritmo, o fato do pivô ser uma mediana de três números complicou a implementação do mesmo.

É importante destacar que ainda existe diversas oportunidades para melhoria, uma refatoração do código poderia lidar com a quantidade excessiva de arquivos que são gerados, otimizar o algoritmo do quicksort e uma melhor organização do código.

- Referências

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Instruções de compilação e execução

Compilação:

No Linux, abra, pelo terminal, o diretório que contém o arquivo Makefile e digite o comando make. Tal comando irá criar o programa juntamente com arquivos necessários para a compilação.

Execução:

Para a execução do código digite no terminal.
<diretório do programa>./bin/programa <par_1> <par_2> <par_3> <par_4> <par_5>

<par_1> → deve ser o nome do arquivo de entrada, parâmetro obrigatório.
<par_2> → deve ser o nome do arquivo de saída, parâmetro obrigatório.
<par_3> → deve ser o numero de entidades a serem lidas por rodada, parâmetro obrigatório.
<par_3> → deve ser o numero de fitas, parâmetro obrigatório.
<par_2> → deve ser o nome do arquivo de log, utilizado pela biblioteca memlog, parâmetro opcional.

Os parâmetros devem ser inseridos nesta em ordem, para que o programa funcione corretamente.