Trabalho Prático 1
Múltiplas Ordenações

Igor Henrique Martins de Almeida
2023028536

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

igorhma@ufmg.br

1. Introdução

O problema proposto foi implementar uma estrutura de dados que, a partir de uma base de dados xCSV lida de um arquivo, geraria 3 listas ordenadas, onde a chave de ordenação de cada lista deveria atender a um critério, por exemplo: name, id e address.

2. Método

O programa foi desenvolvido na linguagem C, compilado pelo compilador GCC da GNU Compiler Collection.

2.1. Estrutura de Dados

A implementação do programa teve como base um struct 'OrdInd' utilizado para armazenar os dados disponibilizados no arquivo xCSV. A struct apresenta 3 atributos inteiros responsáveis por:

1 - armazenar o número de registros contidos no arquivos;

2 - armazenar o número de atributos;

3 - armazenar o tamanho do payload.

Também estava presente na struct 5 ponteiros para ponteiros de char que tinham como responsabilidade armazenar os dados que seriam carregados do arquivo, como, por exemplo:

1 - atributos;

2 - nomes;

3 - ids;

4 - endereços;

5 - payloads.

Durante a leitura do arquivo cada um dos atributos eram armazenados com um índice que seria responsável para a ordenação de acordo com a especificação. Para que desse certo, os atributos que tivessem em uma mesma linha sempre eram armazenados na struct com um mesmo índice, ou seja, para a primeira linha contendo informações a serem ordenadas, os atributos seriam armazenados com índice 0, para a segunda, com índice 1 e assim por diante.

2.2. TADs

Com o intuito de modularizar o código, foram criados dois arquivos de cabeçalho (.h).

O primeiro deles, 'ordenacao.h', ficou responsável pela criação, destruição e atribuição de valores da struct. Também era responsável por carregar o arquivo que seria passado como parâmetro e retornar o número de atributos, o número de linhas e o tamanho do payload, assim como os atributos presentes nos arquivos.

O segundo deles, 'algoritmos.h', foi responsável pela implementação dos 3 algoritmos escolhidos: BubbleSort, QuickSort e InsertionSort. Também continha a implementação de outras funções que são usadas por esses algoritmos durante a ordenação, como o Swap e a Partição.

2.3. Funções

Dentro dos dois arquivos criados, foram pensados a implementação das seguintes funções:

- void Destroi(OrdInd_ptr o): recebia um ponteiro para a struct e liberava a memória alocada para ela.
- OrdInd_ptr Cria(): responsável pela alocação de memória suficiente para uma struct OrdInd e pela inicialização dos valores dentro dela.
- int CarregaArquivo(OrdInd_ptr o, char *nomeEntrada): recebe uma struct OrdInd e um char com o nome do arquivo. Dentro dessa função era feito a abertura e leitura do arquivo, assim como a alocação de memória para cada atributo e a cópia do conteúdo do arquivo para a memória.

- int NumAtributos(OrdInd_ptr o), int NumLinhas(OrdInd_ptr o), int TamPayload(OrdInd_ptr o): 3 funções que recebiam uma struct e retornavam o número de atributos, número de linhas e o tamanho do payload, respectivamente.
- void Informacoes(OrdInd_ptr o): recebe uma struct e printa na tela os atributos contidos no arquivo.
- void Swap(OrdInd_ptr o, int i, int j): função responsável pela troca de posição dos atributos, recebendo a struct e dois índices que serão trocados. Nessa função todos os atributos eram trocados de posição, dependendo de qual estava sendo utilizado como chave, ou seja, quando uma chave era trocada de posição, todos os outros atributos de mesmo índice também eram trocados.
- int Particao(OrdInd_ptr o, int inicio, int fim, const char *atributo): função utilizada pelo QuickSort e responsável por particionar o vetor.
- void QuickSort(OrdInd_ptr o, int inicio, int fim, const char *atributo): implementação do QuickSort. Recebe a struct, o início e o fim do vetor e a chave que estava sendo ordenada
- void BubbleSort(char **arr, char **arr2, char **arr3, char **arr4, int n): implementação do BubbleSort. Recebe 4 arrays representando os arrays dos atributos e o tamanho total do vetor (número de entradas do arquivo).
- void InsertionSort(OrdInd_ptr ordInd, char **base): implementação do InsertionSort. Recebe a struct e um ponteiro para ponteiro que representa a chave que estava sendo utilizada na ordenação.

3. Análise de Complexidade

3.1. Tempo

Para a análise da complexidade de tempo, levaremos em conta apenas as implementações dos algoritmos, sendo a parte de maior relevância de todo o código, uma vez que, a leitura do arquivo e a alocação de memória são razoavelmente rápidas.

Para cada algoritmo, levaremos em conta um valor L que representa o tamanho médio das strings a serem ordenadas.

Para o BubbleSort, temos que, o algoritmo percorre o array comparando pares adjacentes e realizando trocas quando necessário. Cada elemento será comparado com todos os outros elementos, o que nos leva a $\frac{n^2-n}{2}$ comparações, logo, temos que, em relação à quantidade de comparações, o algoritmo tem complexidade igual a O(n² * L). Em relação às movimentações, temos que, para cada comparação, poderemos ter, no pior dos casos (vetor inversamente ordenado), 3 movimentações, o que nos dá O(n² * L).

Por sua vez, o InsertionSort ordena os elementos deslocando-os para a posição correta. Com isso, temos que, no pior caso, cada elemento irá ser comparado com n-1 elementos, logo, podemos concluir que teremos O(n² * L) comparações. Em relação às movimentações, cada elemento do vetor irá percorrer toda parte já ordenada, pois, no pior caso, o algoritmo sempre entra no loop interno, nos levando a $\frac{2(n-1)+n(n-1)}{2}$ = O(n² * L) movimentações.

Por fim, o QuickSort ordena os elementos pelo princípio de divisão e conquista, ou seja, ele divide o array em um ponto definido pelo pivô e repete o processo recursivamente. Para o melhor caso do algoritmo, aquele onde o array é dividido na metade, teremos que a divisão gerará $T(n) = 2T(\frac{n}{2}) + O(n * L)$, resultado em O(L * n log n). Para o pior caso, aquele onde o array já está ordenado, a divisão equilibrada resultará em $T(n) = T(n-1) + O(n * L)$, o que nos dá O(n² * L).

3.2. Espaço

Para a análise de espaço iremos considerar apenas a função CarregaArquivo e as implementações dos algoritmos, sendo as mais significativas em todo o código. Para CarregaArquivos dependemos da quantidade de atributos ($n_{atributos}$) e da quantidade de registro ($n_{registros}$). Com isso, para essa função, temos um custo O($n_{atributos}$ + $n_{registro}$) = O(n).

Para os algoritmos implementados, temos que, o único que necessita de espaço adicional na memória para a ordenação é o QuickSort, sendo que, no caso médio, temos O(log n) e, no pior caso, temos O(n). Logo, o código irá gastar um espaço equivalente a O(n).

4. Estratégia de Robustez

As principais estratégias utilizadas para uma melhor robustez e segurança foram as diversas validações realizadas por todo o código. Em ‘algoritmos.c’ é feito em diversas funções, por exemplo, a validação de ponteiros nulos para garantia que nenhum endereço de memória nulo será acessado, o que levaria a segmentation fault. Também é realizada a validação dos índices em funções que os utilizam para realizar a ordenação dos atributos, garantindo assim acessos inválidos na memória. Por todos arquivos do código também são disponibilizadas mensagens de erro, para que, em possíveis erros, seja passado para o usuário o que ocasionou o erro e facilitando a correção. Em ‘ordenacao.c’, antes de ser feita a liberação da memória também é realizada uma checagem para garantir que todos os ponteiros estão nulos para uma liberação de memória mais segura.

5. Análise Experimental

Para a análise experimental do código foi utilizado principalmente a ferramenta valgrind, callgrind e gdb. Iremos considerar a saída do callgrind para avaliar a distância de pilha e localidade de referência. Com o callgrind, temos 3 informações principais:

- Instruções (I):
    - I refs: número total de instruções executadas pelo programa.
    - I1 misses: número de vezes que o cache de instruções de nível 1 não conseguiu atender à solicitação e precisou buscar a instrução da memória de nível inferior.
    - LLi misses: número de misses no cache de instruções no último nível.
    - I1 miss rate e LLi miss rate: taxa de misses no cache Il e no último nível, respectivamente.
- Dados (D):
    - D refs: total de acessos a dados (leitura e escrita).
    - D1 misses: número de vezes que o cache de dados de nível 1 falhou em atender à solicitação.
    - LLd misses: misses no cache de dados no último nível.
    - D1 miss rate e LLd miss rate: taxa de misses no cache D1 e no último nível, respectivamente.

- Cache de Último Nível (LL):
    - LL refs: número total de referências ao cache de último nível.
    - LL misses: total de misses no cache de último nível.
    - LL miss rate: taxa de misses no cache de último nível.

Para cada algoritmo foram realizados testes com arquivos de tamanhos diferentes, sendo eles: 1000 registros e payload de tamanho 1000, 1000 registros e payload de tamanho 5000, 5000 registros e payload de tamanho 5000.

5.1. QuickSort

	r1000.p1000	**r1000.p5000**	**r5000.p5000**
I refs	62.662.419	274.113.157	1.378.609.041
I1 misses	1.644	1.623	1.786
LLi misses	1.623	1.605	1.778
D refs	22.876.141	99.729.123	501.689.939
D1 misses	114.267	355.877	1.899.946
LLd misses	20.588	83.090	1.171.495
LL refs	115.911	357.500	1.901.732
LL misses	22.211	84.695	1.173.273

Percebe-se que, com o aumento do tamanho de registros e do tamanho do payload, as instruções executadas e o acesso a dados aumenta consideravelmente, o que era de se esperar. Porém, apesar dos misses também aumentarem consideravelmente com o aumento da quantidade de registros, a taxa de erro permanece por volta de 0,5%. Com isso, conclui-se que o programa está acessando instruções e dados próximos uns dos outros na memória. Isso ocorre porque o cache armazena blocos de memória (linhas de cache) em vez de bytes individuais, e o baixo número de misses indica que os blocos carregados contêm dados úteis para acessos subsequentes.

5.2. BubbleSort

	r1000.p1000	r1000.p5000	r5000.p5000
I refs	213.040.855	423.933.049	5.237.757.451
I1 misses	1.636	1.636	1.690
LLi misses	1.601	1.601	1.684
D refs	87.731.969	164.194.467	2.163.594.488
D1 misses	1.864.469	2.102.720	56.193.634
LLd misses	20.229	82.739	1.169.333
LL refs	1.866.105	2.104.356	56.195.324
LL misses	21.830	84.340	1.171.017

Com o BubbleSort, percebemos um comportamento semelhante ao QuickSort, sendo que, para arquivos com a mesma quantidade de registros porém com payloads diferentes, há um aumento nas instruções e no acesso de dados, mas o aumento mais significativo ocorre quando há o aumento no número de registros. Porém, no BubbleSort, percebe-se um aumento significativo na quantidade de misses, principalmente na leitura e escrita de dados em cache de nível 1 (D1). Em um arquivo com 5000 registros e payload de tamanho 5000 temos uma taxa de 2,6% de miss, o que nos leva a concluir que a distância de pilha tende a ser maior, ou seja, instruções e dados utilizados recentemente acabam sendo expulsos do cache antes de serem utilizados novamente.

5.3. InsertionSort

	r1000.p1000	r1000.p5000	r5000.p5000
I refs	205.395.168	415.493.333	5.011.991.581
I1 misses	1.635	1.635	1.686
LLi misses	1.600	1.600	1.679
D refs	99.485.827	175.534.566	2.438.385.901
D1 misses	679.307	921.796	28.422.925
LLd misses	20.229	82.739	1.169.346

LL refs	680.942	923.431	28.424.611
LL misses	21.829	84.339	1.171.025

Como visto nos outros dois algoritmos, o InsertionSort se comporta de forma parecida em relação ao aumento de instruções e acesso a dados, sendo um aumento pouco significativo quando só se altera o tamanho do payload e um aumento muito significativo quando se há o aumento da quantidade de registros. Porém, esse algoritmo tende a ter uma melhor eficácia em relação a distância de pilha em relação ao BubbleSort porém ainda pior do que em relação ao QuickSort. Temos que, para 1000 registros, a taxa de miss relacionada a cache de nível 1 se mantém menor que 1%, indicando uma distância de pilha pequena, indicando que muitos acessos a dados também se beneficiam da reutilização de valores que já estão no cache. Contudo, para 5000 registros, a taxa de miss em D1 já sobe para 1,2%, indicando um pior manejo da cache em relação aos outros testes.

6. Conclusões

Após a implementação do programa, nota-se duas partes distintas para o desenvolvimento do código. A primeira, sendo a implementação da estrutura de dados da forma correta e funcional para que fosse possível uma melhor fluidez no decorrer da segunda parte. Essa segunda parte tinha como objetivo utilizar a estrutura implementada anteriormente em conjunto com os algoritmos de ordenação para que fosse realizado a ordenação correta dos dados disponibilizados nos arquivos. Havia, também, um pequeno desafio, durante a primeira parte, que seria a leitura e alocação de memória correta para os dados do arquivo.

O principal esforço se deu principalmente durante o desenvolvimento da primeira parte, uma vez que a estrutura de dados é a parte indispensável do trabalho, sendo ela indispensável pelo armazenamento dos dados que seriam posteriormente ordenados. Como já era de conhecimento prévio a implementação e funcionalidade dos algoritmos escolhidos, o principal desafio da segunda parte foi apenas em relação ao tamanho das entradas, pois poderia ocasionar grandes gargalos na execução do programa.

Ao fim do trabalho, fica evidente a importância dos testes utilizando-se a ferramenta valgrind e suas diversas funcionalidades para um melhor entendimento do uso da memória pelo programa. Essa prática nos leva a um desenvolvimento mais consciente e produtivo, fazendo com que possamos entender melhor como deixar um código mais eficaz.

7. Bibliografia

SEWARD, Julian et al. Valgrind Documentation. 3.24.0. [S. l.], 31 out. 2024. Disponível em: https://valgrind.org/docs/. Acesso em: 8 dez. 2024.

STALLMAN, Richard; PESCH, Roland; SHEBS, Stan, et al. Debugging with gdb: The gnu Source-Level Debugger. Free Software Foundation: [s. n.], 2017?. ISBN 978-0-9831592-3-0.