Trabalho Prático 2 de Estrutura de Dados

Roberth Alves Parreiras

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brazil

roberthparreiras@ufmg.br

1. Introdução

O problema a ser tratado nesse documento é acerca da implementação de quatro configurações de ordenação. Como ainda há roubo de consciências, a Rellocator CO. solicitou implementar quatro configurações de ordenação para tentar minimizar, ainda mais, os roubos. Implementando esses métodos, deseja-se responder duas hipóteses: a primeira é sobre se irá manter a configuração desejada após as informações serem ordenadas; e a segunda é sobre se os métodos de ordenação irão causar impacto no desempenho dos servidores de envio.

Para resolver esse problema, foi implementado um arquivo de código(Main.cc), junto ao Makefile. Com relação aos comandos, eles são lidos do arquivo na Main.cc e são executados seus três parâmetros no formato de *String*, como vai ser descrito no apêndice.

O documento é divido em **2. Implementação**, **3. Análise de complexidade**, **4. Configuração Experimental, 5. Resultado, 6. Conclusão, Apêndice: Instruções de Compilação e Execução .**

2. Implementação

A Estrutura de Dados do código é composta dos métodos *HeapSort, QuickSort, Juntar, MergeSort1, MergeSortDefinitivo, RadixSort*. Já funções, foram implementadas *BinaryToDecimal,* e a própria *main*.

HeapSort – ele compara os nós pais com os nós filhos. Se o filho é maior que o pai, troca-se as posições entre eles. Faz essa checagem até o tamanho do filho(((tamanho do vetor) / 2) * 2 + 1) ser maior que o tamanho do vetor. Ele não foi dividido em vários métodos, tudo é feito dentro do próprio método HeapSort.

QuickSort – ele coloca os valores menores a esquerda do vetor, e os maiores a direita do vetor. Como ele é divisão e conquista, faz essa divisão para todo o vetor. Ele foi implementado de forma recursiva.

Juntar – como o próprio nome diz, ele copia os nomes e dados para um vetor auxiliar, compara para ver se estão ordenados ou não, e depois essa copia volta para os vetores originais. Utilizado para o *MergeSort.*

MergeSort1 – divide ao meio o vetor, e depois chama recursivamente do início ao meio, e do meio ao final. Chama-se também o método *Juntar* para todo o vetor.

MergeSortDefinitivo – ele cria os vetores auxiliares com memória dinâmica e chama o método MergeSort1.

RadixSort – Transformando o binário em decimal, pela função *BinaryToDecimal,* ele compara os valores convertidos para inteiro, e depois converte novamente para binário.

BinaryToDecimal – transforma binário para decimal.

main – recebe os dados do arquivo e chama os métodos supracitados.

A configuração utilizada para testar o programa foi:
- Windows 10 Home;
- Linguagem C++;
- Compilador ubuntu0.18.04.1;
- Editor replit;
- Processador Intel core i5-8265 1.60 GHz;
- Mémoria RAM 8 GB;

3. Análise de complexidade

. HeapSort – complexidade de tempo: esse método realiza um primeiro loop para trocar alguns valores de posição, tendo sua complexidade $O(n)$. Depois em outro loop, faz a comparação entre pais e filhos, caracterizando como $O(nlogn)$. Como a entrada não interfere em seu tempo, pior e melhor caso tem sua complexidade assintótica de tempo $\Theta(nlogn)$.

. HeapSort – complexidade de espaço: esse método utiliza heap auxiliar de tamanho $O(n)$. Portanto, a complexidade assintótica de espaço desses métodos é $\Theta(n)$.

. QuickSort – complexidade de tempo: esse método realiza um primeiro loop contendo dois loops internos que, se o pivô escolhido for os extremos do vetor, cai em seu pior caso, que é $O(n^2)$. Já seu melhor caso, quando é escolhido o pivô de forma que divide igualmente o vetor, tem sua complexidade $O(nlogn)$. Dessa forma, sua complexidade assintótica de tempo para o pior e melhor caso é, respectivamente, $\Theta(n^2)$ e $\Theta(nlogn)$.

. QuickSort – complexidade de espaço: esse método cria uma pilha de execução que em seu pior caso é $O(n)$ e seu melhor caso é $O(\log_2 n)$. Portanto, a complexidade assintótica de espaço no pior e melhor caso é, respectivamente, $\Theta(n)$ e $\Theta(\log_2 n)$.

. Juntar – complexidade de tempo: esse método contém dois loops que fazem operações constantes e que tem sua complexidade $O(n)$. A complexidade assintótica de tempo desse método é $\Theta(n)$.

. Juntar – complexidade de espaço: esse método utiliza estruturas auxiliares unitárias que tem sua complexidade $O(1)$. Portanto, a complexidade assintótica de espaço desses métodos é $\Theta(1)$.

. MergeSort1 – complexidade de tempo: esse método chama ele mesmo de forma recursiva, sendo essa chamada de complexidade $O(logn)$. Como ela chama o método *Juntar*, que tem complexidade de tempo $O(n)$, sua complexidade assintótica de tempo é $\Theta(nlogn)$.

. MergeSort1 – complexidade de espaço: esse método utiliza estruturas auxiliares unitárias que tem sua complexidade $O(1)$. Portanto, a complexidade assintótica de espaço desses métodos é $\Theta(1)$.

. MergeSortDefinitivo – complexidade de tempo: esse método chama *MergeSort1*, que tem sua complexidade de tempo $O(nlogn)$. Portanto, a complexidade assintótica de tempo desses método é $\Theta(nlogn)$.

. MergeSortDefinitivo – complexidade de espaço: esse método utiliza estruturas auxiliares lineares que tem complexidade de espaço $O(n)$. Portanto, a complexidade assintótica de espaço é $\Theta(n)$.

. BinaryToDecimal – complexidade de tempo: essa função tem um loop que roda numa entrada de tamanho n, com complexidade de tempo sendo $O(n)$. Portanto, a complexidade assintótica de tempo é $\Theta(n)$.

. BinaryToDecimal – complexidade de espaço: esse método utiliza estruturas auxiliares unitárias que tem complexidade de espaço $O(1)$. Portanto, a complexidade assintótica de espaço é $\Theta(1)$.

. RadixSort – complexidade de tempo: esse método tem um primeiro loop que transforma *string* em inteiro, sendo sua complexidade de tempo $O(n)$. Depois, vem o loop que faz todo a sua ordenação, tendo sua complexidade de tempo $O(n\kappa)$, sendo κ o tamanho da *string*. Logo, a complexidade assintótica de tempo é $\Theta(n\kappa)$.

. RadixSort – complexidade de espaço: esse método utiliza estruturas auxiliares lineares que tem complexidade de espaço $O(n + s)$, sendo s o tamanho do alfabeto. Portanto, a complexidade assintótica de espaço é $\Theta(n + s)$.

. main – complexidade de tempo: essa função tem um loop que lê dados do argumento, sendo de tamanho n, como também chama os métodos supracitados. Dessa forma, a complexidade assintótica do seu pior e melhor caso é, respectivamente, $\Theta(n^2)$ e $\Theta(nlogn)$.

. main – complexidade de espaço: essa função utiliza estruturas auxiliares lineares que tem complexidade de espaço do pior e melhor caso sendo, respectivamente, $O(n + s)$ e $O(n)$. Portanto, a complexidade assintótica de espaço do pior e do melhor caso é, respectivamente, $\Theta(n + s)$ e $\Theta(n)$.

A complexidade assintótica de tempo no pior caso geral é $\Theta(n^2)$, devido ao *QuickSort* na função **main**. Já a complexidade assintótica de espaço no pior caso geral é $\Theta(n + s)$, devido ao *RadixSort* na função **main**.

4. Configuração Experimental

Foi implementado quatro configurações para ordenar os dados do arquivo de entrada, que varia entre 1000, 10000, 50000, 100000 e 200000 linhas. Será passado como argumento para a função *main* qual o tamanho desejado, como também qual das configurações quer que ordene, sendo elas:

. Configuração 1: primeiro ordena em relação aos dados em binário, com o *HeapSort,* logo após, ordena em relação aos nomes, utilizando o *QuickSort.*

. Configuração 2: primeiro ordena em relação aos dados em binário, com o *RadixSort,* logo após, ordena em relação aos nomes, utilizando o *QuickSort.*

. Configuração 3: primeiro ordena em relação aos dados em binário, com o *HeapSort,* logo após, ordena em relação aos nomes, utilizando o *MergeSort.*

. Configuração 4: primeiro ordena em relação aos dados em binário, com o *RadixSort,* logo após, ordena em relação aos nomes, utilizando o *MergeSort.*

5. Resultado

Para ser medido o tempo de execução, foi utilizado a biblioteca *chrono*, com o comando *std::chrono::high_resolution_clock::now()*, antes das configurações de ordenação, e depois com o comando *std::chrono::duration_cast<std::chrono::milliseconds>(resultado).count()*, para descobrir quanto foi a medida em milissegundos.

Para 1000 linhas, obteve os seguintes tempos de execução (em milissegundos):

Mínimo: 1ms Máximo: 2ms Média: 1,75ms

Para 10000 linhas, obteve os seguintes tempos de execução (em milissegundos):

Mínimo: 22ms Máximo: 34ms Média: 27,75ms

Para 50000 linhas, obteve os seguintes tempos de execução (em milissegundos):

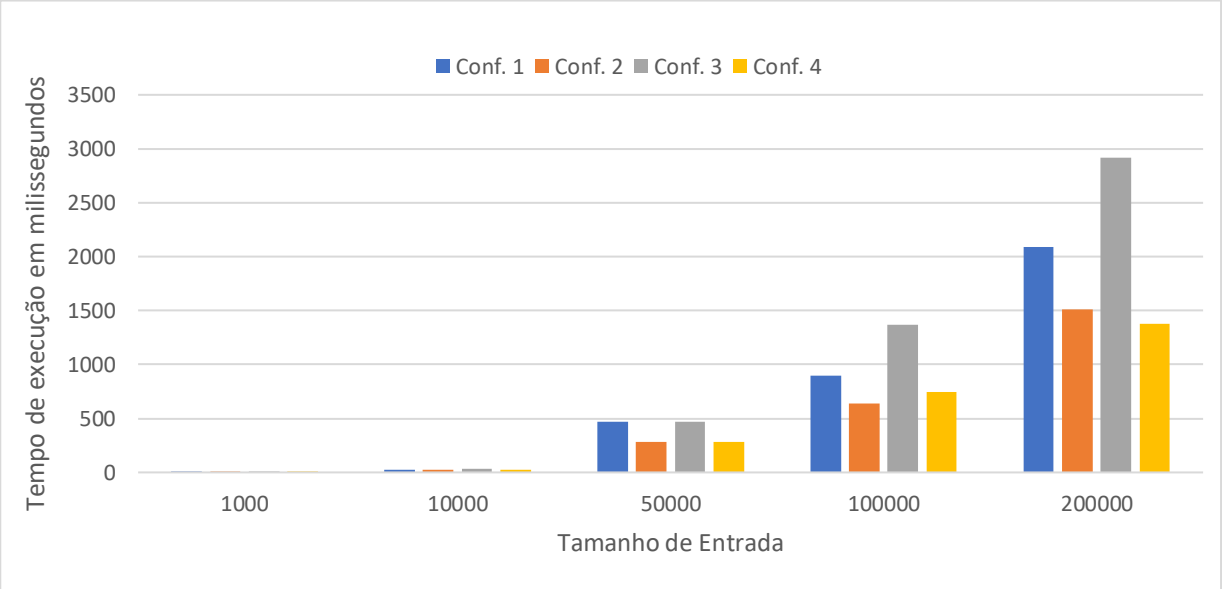
Mínimo: 279ms Máximo: 470ms Média: 375ms

Para 100000 linhas, obteve os seguintes tempos de execução (em milissegundos):

Mínimo: 747ms Máximo: 1366ms Média: 912ms

Para 200000 linhas, obteve os seguintes tempos de execução (em milissegundos):

Mínimo: 1377ms Máximo: 2917ms Média: 1973,25ms



Com relação a estabilidade das configurações, é notório que os métodos *HeapSort*, *QuickSort* e *MergeSort*, separados, são não estáveis. Já o *RadixSort* é estável. Entretanto, quando se chama as configurações, percebe-se que são todas não estáveis, ocasionando, assim, com que a saída dos binários não esteja ordenada. Um ponto a se observar é que as configurações 3 e 4 têm seus binários ordenados de ordem decrescente.

6. Conclusão

Para minimizar ainda mais o problema de roubo de consciências, foi proposto a implementação de algoritmos de ordenação que são divididos em quatro configurações. Implementando-os, deve-se

responder duas hipóteses: a primeira se irá manter a configuração desejada após as informações serem ordenadas; e a segunda é sobre se os métodos de ordenação irão causar impacto no desempenho dos servidores de envio.

Foi visto que todas as configurações propostas, nenhuma conseguiu satisfazer a primeira hipótese, sendo todas não estáveis. Com relação a segunda hipótese, como foi visto no capítulo anterior, para arquivos com até 50000 linhas, não existe uma diferença muito grande entre todas as configurações. Porém, a partir de 100000 linhas, as configurações 1 e 3 começam a ter uma diferença considerável com relação as configurações 2 e 4. Entre 2 e 4, percebe-se que a configuração 4 se sai melhor que a outra. Sendo assim, a possível configuração a se implementar em definitivo é a configuração 4: *RadixSort* e *MergeSort*. No entanto, essa diferença não é tão relevante considerando esses tamanhos do arquivo de entrada.

Para ser implementado os métodos, houveram várias dificuldades a serem enfrentadas, tais como implementar o *RadixSort* para ordenar os dados em binário, que foi resolvido transformando o binário em inteiro. Houve também o problema de *Segmentation Fault (core dumped)*, que foi resolvido usando alocação dinâmica de espaço.

Com esse trabalho prático, deu para aprender sobre complexidade de métodos de ordenação, como também sobre a estabilidade dos mesmos.

Referencias

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.

https://pt.wikipedia.org/wiki/Heapsort#Implementa%C3%A7%C3%A3o_em_C

https://pt.wikipedia.org/wiki/Quicksort

https://www.programiz.com/c-programming/examples/binary-decimal-convert

https://pt.wikipedia.org/wiki/Radix_sort#C%C3%B3digo_em_C

https://peter.bloomfield.online/convert-a-number-to-a-binary-string-and-back-in-cpp/

https://pt.stackoverflow.com/questions/278269/como-que-eu-posso-medir-o-tempo-de-execu%C3%A7%C3%A3o-de-um-programa-em-c

Apêndice

Instruções de Compilação e Execução

Para compilar você deve seguir os seguintes passos:

. Utilizando um terminal, execute o Makefile digitando make, e após compilar, utilize o seguinte comando:

./bin/run.out [nome do arquivo] [configuração] [quantidade de linhas] ;

. Com esse comando, deve ocorrer a saída correta, seguindo as especificações contidas no arquivo.