Documentação - Trabalho Prático 2

Gabriel Medeiros Teixeira
Matrícula: 2020054420
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
Sistemas de Informação
Estruturas de Dados - 2021/1

gabrielmt09@ufmg.br

1. Introdução

O problema proposto foi implementar algoritmos de ordenação a fim de e ordenar nomes e números binários, que representam consciências no sistema de "upload de mentes" da Rellocator CO., a fim de estudar a eficácia de certas combinações desses métodos, visando uma otimização do processo realizado pela empresa que foi citado anteriormente. Para tal, foram implementados os métodos de ordenação **Quicksort**, **Mergesort**, **Heapsort** e **Radix Exchange Sort**, com Quicksort e Mergesort sendo usados para ordenar nomes e Heapsort e Radix Exchange Sort para dados binários. O objetivo final desejado com uma ordenação completa (um método ordena os nomes e outro os binários) é obter os nomes, inseridos na entrada, alfabeticamente ordenados, tendo nomes iguais ordenados entre si pelos números binários que representam suas consciências, em ordem crescente.

2. Estruturas de dados

Os métodos de ordenação foram implementados por meio da utilização de namespaces, um para cada método, cujas assinaturas foram definidas na pasta “include”, nos arquivos “heapSort.h”, “mergeSort.h”, “quickSort.h” e “radixSort.h”, para então serem implementados na pasta “src”, nos arquivos “heapSort.cpp”, “mergeSort.cpp”, “quickSort.cpp” e “radixSort.cpp”.

2.1. Namespaces

2.1.1. mergeSort

Namespace que contém as funções que constituem o método de ordenação Merge Sort. Tais funções são “merge” e “mergeSort”. que serão detalhadas na seção 2.2.

2.1.2. quickSort

Namespace que contém as funções que constituem o método de ordenação Quick Sort. Tais funções são “ordena”, “particao” e “quickSort”.

2.1.3. heapSort

Namespace que contém as funções que constituem o método de ordenação Heap Sort. Tais funções são “refaz”, “constroi” e “heapSort”.

2.1.4. radixSort

Namespace que contém as funções que constituem o método de ordenação Radix Sort. Tais funções são “sort” e “radixSort”.

2.2. Funções

2.2.1. mergeSort::merge

A função recebe como parâmetros os arrays de string “nomes” e “dados” (que armazenam os nomes e dados inseridos na entrada, respectivamente, um por posição), além dos inteiros “esq”, “meio” e “dir”, que indicam as posições inicial, central e final dos arrays. “merge” é usado para ordenar os dois arrays passados como parâmetro com base em “nomes”, com as únicas modificações feitas em “dados” sendo as exatas mesmas alterações de índice que ocorrem para a ordenação de “nomes” (“nomes[x]” deve referir-se a “dados[x]”). A função presume que os itens das posições de “esq” até “meio” e de “meio + 1” até “dir” estão ordenados alfabeticamente entre si, para então utilizar de dois arrays auxiliares a fim de concluir a ordenação de “nomes” como um todo, através da comparação entre as duas partes já ordenadas.

2.2.2. mergeSort::mergeSort

Representa o método de ordenação **Merge Sort**. Recebe como parâmetros os arrays de string “nomes” e “dados”, além de “comeco” e “fim”, inteiros que indicam a posição inicial e final dos arrays. Executa duas chamadas recursivas, passando os índices equivalentes à primeira metade dos arrays para uma e à segunda para a outra. Essas chamadas recursivas permitem a separação das partes dos arrays até itens unitários. Em seguida a função “merge” é executada para cada chamada recursiva de “mergeSort”, permitindo a reconstrução ordenada dos arrays (com base no array “nomes”).

2.2.3. quickSort::particao

Recebe como parâmetros os arrays de string “nomes” e “dados”, além dos inteiros “esq” (primeira posição dos arrays) e “dir” (última posição dos arrays) e dos ponteiros de inteiro “i” e “j”. Reorganiza o array “nomes” de forma que todos os elementos à esquerda do pivô, que era o item central antes da ordenação, sejam menores que ele e à direita sejam maiores. O valor apontado por “i” passa a indicar o primeiro elemento à direita desse pivô e “j”, o primeiro à esquerda. Realiza em “dados” as mesmas mudanças de índice que faz em “nomes”.

2.2.4. quickSort::ordena

Recebe como parâmetros os arrays de string “nomes” e “dados”, além dos inteiros “esq” (primeira posição dos arrays) e “dir” (última posição dos arrays). Utiliza da função “particao” e duas chamadas recursivas de “ordena” (uma recebe os índices referentes aos elementos da esquerda do pivô na chamada de “particao” e a outra, os índices referentes aos da direita). A partir dessa estratégia, ordena “nomes” em ordem alfabética e realiza as mesmas alterações de índice em “dados”.

2.2.5. quickSort::quickSort

Representa o método de ordenação **Quick Sort**. Recebe como parâmetros os arrays de string “nomes” e “dados”, além do inteiro “n”, que indica o tamanho dos arrays. Ordena “nomes” em ordem alfabética e realiza alterações de índice correspondentes em “dados” a partir da chamada de “ordena”.

2.2.6. heapSort::refaz

Recebe como parâmetros os arrays de string “nomes” e “dados”, além dos inteiros “esq” (primeira posição dos arrays) e “dir” (última posição dos arrays). Verifica se “dados[esq]”, raiz dessa subárvore, é maior que seus filhos (“dados[esq * 2 + 1]” e “dados[esq * 2 + 2]”), a fim de colocar o conteúdo dessa raiz na posição correta na perspectiva de um heap. Caso não seja maior, as trocas necessárias para que a raiz seja maior que todos os filhos são feitas. Toda mudança de posição efetuada em “dados” têm alterações equivalentes feitas em “nomes”.

2.2.7. heapSort::constroi

Recebe como parâmetros os arrays de string “nomes” e “dados”, além do inteiro “n”, que representa o tamanho dos arrays. Utiliza da função “refaz” para transformar “dados” em um heap, ao chamá-la definindo como posição inicial a primeira que tem um filho, da direita para a esquerda, (n/2), para então repetir o processo para a posição anterior e assim por diante, até executar “refaz” com “dados[0]” como a primeira posição.

2.2.8. heapSort::heapSort

Representa o método de ordenação **Heap Sort**. Recebe como parâmetros os arrays de string “nomes” e “dados”, além do inteiro “n”, que representa o tamanho dos arrays. Utiliza a função “constroi” para transformar “dados “ em um heap. A partir desse ponto, passa a raiz para a última posição e reconstrói o heap, usando “refaz”, não levando em conta a raiz que acabou de ser movida. Esse processo é repetido n - 1 vezes até que o array passado para refaz seja equivalente às 2 primeiras posições de “dados”. Assim que essa última chamada é concluída, a ordenação também está. Vale lembrar que todas as alterações de índice feitas em “dados” também são realizadas em “nomes”.

2.2.9. radixSort::sort

Recebe como parâmetros os arrays de string “nomes” e “dados”, além dos inteiros “esq” (primeira posição dos arrays, “dir” (última posição) e “indiceBit” (a posição do bit no número binário que está comparando). Realiza uma ordenação como o “Quicksort” bit a bit, ordenando os itens com base no array “dados”, do primeiro bit da esquerda (mais significativo) até o último da direita. A partição resultante da esquerda contém os elementos cujo bit da iteração atual (indicado por “indiceBit”) é 0 e, a da direita, àqueles em que o bit é 1. São realizadas chamadas recursivas de “sort” para cada partição, com “indiceBit” selecionando o bit seguinte. O resultado é “dados” em ordem crescente. Todas alterações de índice feitas em “dados” também são realizadas em “nomes”.

2.2.10. radixSort::radixSort

Representa o método de ordenação **Radix Sort**. Recebe como parâmetros os arrays de string “nomes” e “dados”, além dos inteiros “esq” (primeira posição dos arrays e “dir” (última posição). Ordena o array binário “dados” em ordem crescente por meio da chamada "sort".

2.2.10. Função *main*

É a principal função do programa, recebendo como alguns de seus parâmetros o arquivo de entrada “homologacao.txt”, a configuração desejada e o número de linhas dessa entrada que devem ser consideradas. Ela utiliza a função “getline” para ler cada linha do arquivo até o limite inserido na entrada, armazenando o nome e o dado na n-ésima linha da entrada nas n-ésimas posições dos arrays de string “nomes” e “dados”, respectivamente.

Por meio de um “switch”, seleciona a configuração desejada para o programa (1, 2, 3 ou 4). Cada configuração chama dois métodos de ordenação, em que um (o primeiro a ser chamado) ordena o conteúdo das linhas da entrada de forma crescente por meio dos dados binários e o outro realiza uma segunda ordenação, sobre o resultado do primeiro, levando em consideração a ordem alfabética dos nomes. Caso uma configuração não existente seja selecionada, um erro é lançado.

Quanto a saída, é realizado um “for” que junta novamente os pares de nomes e dados, imprimindo o conteúdo das linhas especificadas na entrada após a ordenação.

3. Análise de Complexidade

3.1. mergeSort

- Considere n como o tamanho do array “nomes”.

3.1.1. Tempo

A complexidade de tempo do método é constante para qualquer entrada, não existindo melhor ou pior caso. Assim, tal complexidade, referente ao número de operações, é O(n log(n)).

3.1.2. Espaço

A complexidade de espaço adicional do método é O(n), espaço esse que é necessário para a alocação dos vetores de string que são utilizados para o particionamento dos vetores “nomes” e “dados”, que devem ser ordenados nas chamadas de “merge”.

3.2. quickSort

3.2.1. Tempo

- Considere n como o tamanho do array “nomes”.

Melhor caso: O melhor caso ocorre quando cada partição divide o conjunto considerado do array “nomes” em duas partes iguais. Assim, a complexidade de tempo, referente ao número de operações, é O(n log(n)).

Pior caso: Ocorre quando o pivô escolhido for maior ou menor elemento do array “nomes”, parâmetro da função “quickSort”, que representa esse método de ordenação. Nesse caso é ordenado um elemento por vez. Assim, a complexidade de tempo referente ao número de ordenações é $O(n^2)$.

3.2.2. Espaço

A complexidade de espaço do método depende puramente da quantidade de chamadas recursivas de Ordena, sendo assim, é O(log(n)).

3.3. heapSort

- Considere n como o tamanho do array “dados”.

3.3.1. Tempo

O método chama a função “constroi”, que por sua vez chama n/2 vezes o método “refaz”, com custo “O(log(n))”, além de executar “refaz” n-1 vezes em outro loop. Assim, a complexidade de tempo do método, com relação ao número de operações realizadas, é O(n log (n)).

3.3.2. Espaço

O método não requer uso de memória auxiliar, portanto, sua complexidade de espaço é O(1).

3.4. radixSort

- Considere n o número de elementos do array “dados” e k o número de bits de cada número binário armazenado nesse vetor.

3.4.1. Tempo

O método faz k passagens pelo vetor de n elementos, sendo assim, a complexidade de tempo referente ao número de comparações de bits é O(n * k).

3.4.2. Espaço

Como o método foi implementado utilizando uma adaptação do Quick Sort, a complexidade de espaço é O(log(n)).

3.5. Função main

- Considere n o tamanho dos vetores “nomes” e “dados”, p o tamanho das linhas da entrada e k o tamanho dos números binários representados em cada linha.

3.5.1. Tempo

Melhor caso: Assumindo que log(n) seja maior que k e que n seja maior que p, a complexidade do melhor caso é O(n log(n)).

Pior caso: Ocorre ao chamar o Quick Sort em seu pior caso, possuindo complexidade O(n^2).

3.5.1. Espaço

Considerando que a complexidade de espaço auxiliar necessário para a alocação dos arrays “nomes” e “dados” é O(n), sendo equivalente ao pior caso das configurações de ordenação, a complexidade é O(n).

4. Configuração experimental

Nesta seção serão explicitadas as configurações de métodos de ordenação que foram testadas no programa e as condições em que os experimentos foram realizados. Seguem alguns detalhes de hardware do ambiente de teste:

- Processador: Intel® Core™ i5-9300H CPU @ 2.40GHz
- Memória RAM: 16 GB

O programa foi desenvolvido na linguagem C++ (C++11) e compilado pelo compilador G++ (Ubuntu 9.3.0-17ubuntu1~20.04) da GNU Compiler Collection. Por fim, foi testado em um sistema operacional Windows 10, a partir de um terminal WSL (v1).

As cada configuração experimental do programa é composta de um método de ordenação que ordena as linhas de acordo com os dados binários nelas contidos (Heap Sort ou Radix Sort), em ordem crescente, em seguida de um método que, sobre a primeira ordenação, organiza as linhas em ordem alfabética de acordo com os nomes nelas armazenados (Quick Sort ou Merge Sort).

Os métodos são executados nessa sequência para que a ordenação principal seja referente à ordem alfabética dos nomes e a ordenação entre nomes iguais seja feita com base em seus dados binários. Vale lembrar que, dos quatro métodos mencionados, apenas o Merge Sort é estável. O impacto disso será discutido na seção de resultados.

O tempo de execução de cada uma das configurações foi medido usando a biblioteca “chrono” do c++.

Assim, as configurações usadas são:

Configuração 1:

É usado o Heap Sort (complexidade de tempo O(n log(n))) para ordenação dos dados binários e Quick Sort (complexidade de tempo O(n log(n)) no melhor caso e $O(n^2)$ no pior) para os nomes.

Configuração 2:

É usado o Radix Sort (complexidade de tempo O(n * k)) para ordenação dos dados binários e Quick Sort (complexidade de tempo O(n log(n)) no melhor caso e $O(n^2)$ no pior) para os nomes.

Configuração 3:

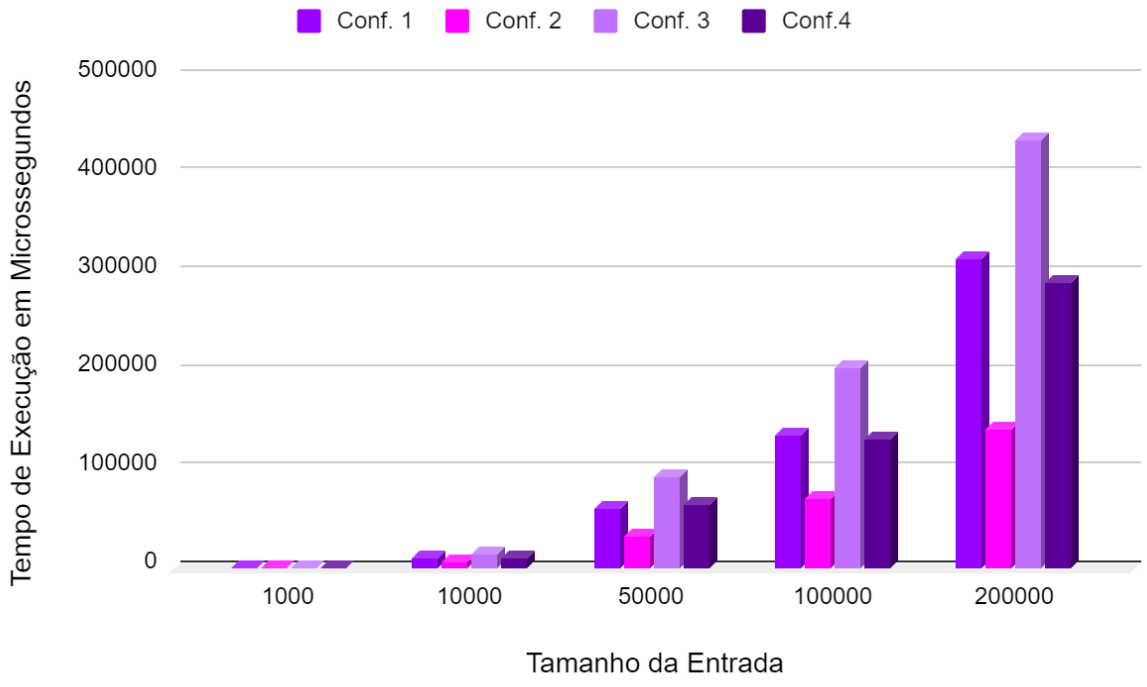
É usado o Heap Sort (complexidade de tempo O(n log(n))) para ordenação dos dados binários e Merge Sort (complexidade de tempo O(n log(n))) para os nomes.

Configuração 4:

É usado o Radix Sort (complexidade de tempo O(n * k)) para ordenação dos dados binários e Merge Sort (complexidade de tempo O(n log(n))) para os nomes.

5. Resultados

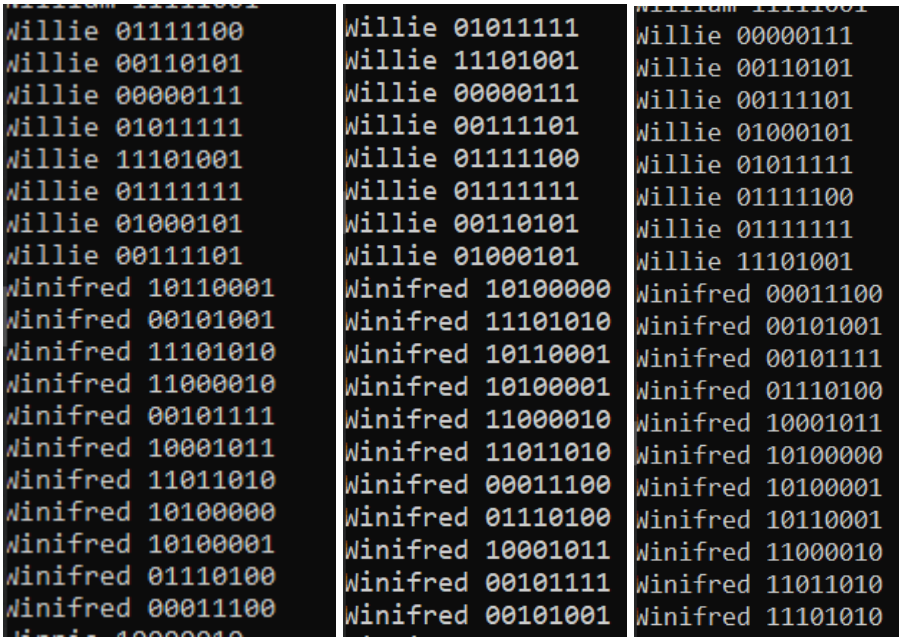
Segue abaixo um gráfico de barras para ilustrar a comparação do tempo de execução das configurações experimentais.



Ao analisar o gráfico, é possível perceber que, quanto ao tempo de execução das configurações utilizadas, a segunda (Radix Sort e Quick Sort) é a mais eficiente em todos os casos (tanto entradas pequenas quanto grandes), sendo a disparidade do tempo de execução desse método proporcional ao tamanho da entrada. Em contrapartida, é notável que a configuração 3 (Merge Sort e Heap Sort) é a menos eficiente, demonstrando um tempo de execução consideravelmente maior do que as outras para todos os tamanhos de entrada. Por fim, as configurações 1 e 4 possuem tempo de execução próximo para todas as entradas, com a 1 sendo ligeiramente mais rápida nos três menores tamanhos de entrada e a 4, nos 2 maiores.

Em conjunto com o mencionado acima, se compararmos as configurações 1 e 3 ou 2 e 4 (em que somente o método usado para ordenar nomes varia), pode-se perceber que, para os testes realizados, o Quick Sort é mais eficiente, no que diz respeito ao tempo de execução, que Merge Sort. Esse comportamento pode indicar que as entradas fornecidas se aproximam do melhor caso do Quick Sort. Já em relação aos métodos usados para ordenar binários, se compararmos as configurações 1 e 2 ou 3 e 4 (em que somente esses métodos variam), podemos perceber que, para as entradas fornecidas, o Radix Sort demonstra-se mais eficiente, no quesito tempo de execução, do que o Heap Sort. Tal comportamento pode indicar que, tendo em vista que a complexidade de tempo do Radix Sort é O(n * k) e a do Heap Sort é (n log(n)) (sendo n a quantidade de linhas a serem ordenadas e k o tamanho dos dados binários de cada linha), para todos os casos testados, log (n) é consideravelmente maior que k.

Nessa perspectiva, apesar de a configuração 2 ser a mais eficiente, no quesito tempo de execução, ela, bem como a 1, não alcança o resultado desejado para a ordenação, como podemos ver no exemplo a seguir:



A imagem da esquerda acima representa um trecho do resultado da configuração 1, a do meio, o mesmo trecho do resultado da 2, e a terceira, das configurações 3 e 4 (resultado ideal).

É perceptível que os resultados das configurações 1 e 2 não mantêm as linhas com nomes iguais ordenadas entre si por seus dados binários em ordem crescente, diferentemente do resultado das configurações 3 e 4. Isso deve-se ao fato de que o Merge Sort é estável e o Quick Sort não, então, como os dados binários são ordenados antes dos nomes, o Quick Sort não fornece o resultado final esperado.

Assim, conclui-se que, levando em conta todos os pontos mencionados acima, a configuração mais eficiente que alcança o resultado almejado é a 4.

6. Conclusão

O desenvolvimento desse projeto tem como objetivo final testar e comparar quatro configurações de métodos para ordenar as “mentes” enviadas para a Rellocator CO., sendo que cada uma é representada por um nome e um número binário de 8 bits. Dessa forma, foram implementados os métodos Heap Sort e Radix Sort para ordenar os binários e Quick Sort e Merge Sort para ordenar os nomes em seguida. Essa implementação foi feita utilizando namespaces, um para cada

método, que continham todas as funções que os compunham, além daquelas que representam os métodos em si. Na pasta final do projeto, há um arquivo “.h” na pasta “include” e um “.cpp” na “src” para cada um deles.

Por fim, foi possível analisar que, apesar de a combinação mais eficiente testada, no quesito tempo de execução, ser a segunda (Quick Sort e Radix Sort), ela não fornece o resultado final desejado pelo fato de o Quick Sort não ser estável, o que resulta em dados binários desordenados entre linhas com o mesmo nome. Assim, conclui-se que a configuração mais eficiente das quatro testadas, que alcança o resultado desejado, é a quarta (Radix Sort e Merge Sort).

7. Referências

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++

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://www.geeksforgeeks.org/measure-execution-time-function-cpp/
https://www.geeksforgeeks.org/chrono-in-c/
https://www.geeksforgeeks.org/merge-sort/
https://www.geeksforgeeks.org/substring-in-cpp/

www.cplusplus.com/reference/string/string

8. Instruções para compilação e execução

Ele deve ser compilado com o Makefile incluído na pasta raiz do programa, usando o comando “make all” no terminal.

Para executar o programa, rode o arquivo “run.out”, na pasta bin, com o arquivo de entrada, configuração desejada (1, 2, 3 ou 4) e número de linhas a serem ordenadas da entrada em questão. Um exemplo (partindo do pressuposto que está na pasta raiz do “TP”) é: “./bin/run.out homologacao.txt 1 50000”.