TP2 - Documentação

Nome: Rodrigo Ferreira Araújo — Matrícula: 2020006990

Agosto 2021

1 Introdução

Esta documentação trata o problema da avaliação temporal e de estabilidade de diferentes configurações de algoritmos para a ordenação conjunta de nomes e dados binários conforme proposto pelo enunciado do trabalho prático.

Nesse sentido, sob o contexto de upload de mentes para corpos sintéticos por meio de servidores da Reallocator CO, o objetivo desta documentação é implementar e testar a velocidade e estabilidade de combinações de algoritmos (4 combinações propostas) para ordenar essas consciências, arranjadas individualmente em um campo "nome" e um campo "dados" (número binário que identifica a consciência).

Para a resolução do problema descrito, foi medido o tempo de execução e analisado a estabilidade de 4 combinações diferentes de algoritmos que ordenam, respectivamente, os dados e os nomes pertencentes às consciências, de modo que é esperado, ao final, que os nomes estejam em ordem alfabética e, para uma sequência de nomes repetidos, os dados estejam em ordem crescente. A linguagem de programação usada é C++.

2 Implementação/Método

2.1 Headers.hpp

Este arquivo contém todos os cabeçalhos das funções implementadas no TP, bem como a definição de um tipo de dado "Registro", que contém apenas dois atributos do tipo string: "dado" e "nome". Esse TAD será usado para armazenar, do arquivo de entrada, os dados e seus respectivos nomes em um array dinâmico, declarado com o tipo "Registro" e com o tamanho passado por argumento no comando de execução. Por fim, note que a nomenclatura das funções seguem um padrão do tipo "[nomefuncao]_[tipo]", em que "tipo" explicita o tipo de dado a ser ordenado, podendo ser "int": dados binários ou "str": nomes (strings).

2.2 Headers.cpp

Este arquivo contém as implementações de todas as funções usadas no programa principal ("main.cpp"), o que inclui as funções necessárias para os algoritmos de ordenação, bem como um método para a impressão dos registros (nomes e dados).

QUICKSORT: O quicksort implementado aqui segue os padrões vistos na aula de Algoritmos de Ordenação do professor Chaimowicz, porém foi adaptado para ordenar arrays de "Registro". Nesse sentido, o procedimento de partição considera como comparação relevante para obtenção dos pivôs a ordem alfabética entre as strings, realizada por meio dos operadores lógicos básicos ($>$, $<$, $=$), que se aplicam ao tipo string nesse contexto.

2.2.1 Particao_str

Cria um pivô x do tipo Registro e particiona o array de Registros em duas seções de modo que, a seção da esquerda terá elementos menores ou iguais ao pivô x, e a seção da direita, elementos maiores ou iguais ao pivô x. Para isso, é criado um Registro w auxiliar para ajustar os vetores.

2.2.2 Ordena_str

Aplica o procedimento Particao_str e realiza chamadas recursivas para completar a ordenação das duas seções do array de Registros por meio dos índices i e j operados dinamicamente.

2.2.3 QuickSort_str

Chama o procedimento Ordena_str com o array completo de Registros e os índices 0 e n-1.

HEAPSORT: De forma semelhante, a implementação do heapsort segue os padrões vistos em aula. O heap é construído e reconstituído no vetor completo de Registros, passado na função principal do heapsort. Note que agora as comparações relevantes são feitas no campo "dados" do Registro original e dos Registros auxiliares criados.

2.2.4 Refaz_int

Função componente do algoritmo heapsort que, no contexto do problema, reconstitui o heap de Registros até que ele esteja ordenado (após sucessivas chamadas com índices menores), comparando dados entre o array de Registros e um Registro x auxiliar.

2.2.5 Constroi_int

Função responsável por construir o heap para posteriormente ser adequado por Refaz_int, usando o array de Registros.

2.2.6 Heapsort_int

Função principal do algoritmo heapsort que constroi o heap usando o array original de Registros e um Registro auxiliar para ordenar o vetor.

RADIX_EXCHANGE_SORT: Para a implementação desse algoritmo, foram seguidos os padrões vistos em aula aliado à uma referência de implementação na internet semelhante para o funcionamento completo do algoritmo, o que inclui a função auxiliar "digit" e o define de "numbits" no arquivo headers.hpp.

2.2.7 digit

Função auxiliar para o método quicksort binário que retorna 1 (verdadeiro) se o bit da posição desejada, por meio de $[numbits - 1 - w]$, onde numbits = 8 (tamnaho da cadeia) e "w" é o parâmetro de controle de acesso.

2.2.8 quicksortB_int

Método de funcionamento semelhante ao quicksort, porém a partição é feita comparando-se bits ao invés de chaves (dados). As chamadas recursivas ordenam as subseções do array de registros "a" pelo bit w-1, caminhando com os índices i e j. Note que nessa função, o campo "dado" é percorrido no seu formato de string original, visando facilitar o acesso e mudança de bits no processo de ordenação.

2.2.9 RadixExchSort_int

Chamada principal da função Radix Exchange Sort, que chama o quicksort binário com o array de Registros completo, com l = 0 e r = N - 1, onde N é o tamanho da entrada, e numbits - 1 = 7, uma vez que o número de bits padrão dos dados é 8.

MERGESORT: Por fim, há a implementação do mergesort conforme o template do algoritmo mostrado em aula. Note que os sub-arrays criados e ordenados na função auxiliar são ambos do tipo Registro, mas as comparações são feitas especificamente no campo "nome".

2.2.10 merge_str

Função auxiliar que realiza a divisão do array dos Registros em dois sub-arrays do tipo Registro alocados dinamicamente,, preenchidos de acordo com o array original do parâmetro, e desalocados ao final da função. Note que, logo após, ocorre a ordenação concomitante com a junção dos sub-arrays, bem como o tratamento do caso em que sobre elementos em um dos sub-arrays.

2.2.11 Mergesort_str

Chamada principal do Mergesort. São realizadas chamadas recursivas e chamadas de merge sob a lógica Pos-Ordem, ou seja, Mergesort(esquerda) -> Mergesort(direita) -> merge.

IMPRIME: Função usada no programa para imprimir todas as informações do array de Registros (nome, dado), conforme o formato da saída esperada.

2.3 main.cpp

Programa diretor do trabalho prático, responsável por ler o arquivo de entrada, armazenar os dados corretamente em um array dinâmico de Registros e executar a ordenação. Ademais, baseado nos valores dos argumentos passados no comando de execução, o programa executa uma das quatro configurações de algoritmos postulados na tabela da descrição do trabalho prático, bem como determina o tamanho da entrada de nomes e dados a

serem ordenadas (linhas do arquivo). Note que há a inclusão da biblioteca "chrono", que será utilizada para calcular precisamente o tempo de execução dos algoritmos, para cada configuração disponível. Por fim, temos a biblioteca "fstream", responsável por comandos de leitura de arquivo.

Observação: Há, no programa, comandos de impressão responsáveis por imprimir o array de Registros antes e depois de cada configuração de algoritmos aliado com uma sinalização que indica qual é qual, tal como por imprimir o tempo de execução em microssegundos. Contudo, haja vista que as impressões causaram instabilidade no cálculo do tempo, os registros de tempo foram feitos com todas as impressões comentadas exceto aquela que mostra o tempo de execução. Contudo, a versão final do Trabalho Prático será entregue com todos os trechos de código de impressão comentados exceto aquele responsável por imprimir os Registros ordenados.

3 Análise de Complexidade

Para a análise de complexidade dessa implementação, especificamente de tempo, consideraremos a comparação de strings (dado e nome) usando $>$, $<$ e $=$ com custo O(1), dado que se trata de um tipo de dado nativo da linguagem C++. Para as notações assintóticas, considere "n" como o número de Registros no array principal.

3.1 Quicksort

Dada a natureza aleatória da ordem dos dados presentes no arquivo de entrada, bem como levando em conta o custo O(1) de comparações entre strings e índices, a complexidade do quicksort implementado segue a complexidade do caso médio típica do algoritmo, que, segundo Sedgewick e Flajolet, é $C(n) = 1.386n \cdot log(n) - 0.846n$, onde n = número de Registros

no array. Portanto, em média, o tempo de execução será próximo da ordem de O($n \cdot log\,(n)$), provocado pela divisão do array em duas subseções de tamanho aproximadamente igual, isto é, n comparações relevantes entre $log\,(n)$ elementos. Para a complexidade de espaço, que envolve tanto o tamanho da entrada como também o tamanho de estruturas auxiliares criadas, nesse contexto, o quicksort apenas trabalha em cima da entrada do array de n Registros e cria uma unidade de Registro auxiliar, que tem custo espacial O(1). Logo, temos uma complexidade de espaço total da ordem O(n).

3.2 Heapsort

O heapsort é um algoritmo especialmente pouco volátil, isto é, mantém a ordem de complexidade relativamente constante independentemente do tamanho da entrada. Com isso, como a ordenação do heap segue a lógica de uma árvore binária completa, as comparações entre strings é feita a partir de nós pais, conferindo se a condição do pai é maior que a dos dois filhos. Portanto, para n (= tamanho da entrada) nós de uma árvore binária completa de altura = $log\,(n)$, temos uma complexidade geral de $C\,(n) = n \cdot log\,(n)$, ou seja, O($n \cdot log\,(n)$). Para a complexidade de espaço, temos uma situação análoga ao quicksort, isto é, além da entrada de n Registros, temos a criação de dois Registros auxiliares na função principal no método Refaz, ambos com custo O(1). Desse modo, temos uma complexidade de espaço total da ordem O(n).

3.3 Radix Exchange Sort

Para o Radix Exchange Sort, como visto nas aulas, a complexidade dependerá do custo de extração dos bits das chaves. Nesse sentido, considerando custo O(1) para acesso à posições específicas na string de bits, temos custo $C(n) = n \cdot k$, onde n é o tamanho da entrada e k é o tamanho da string de bits (campo "dado"), que, nesse caso, são 8 bits. Desse modo, $C(n) = n \cdot 8$ e, portanto, O(n), considerando que o tamanho da entrada aumenta expressivamente. Já a complexidade de espaço é de O(n), já que é criado apenas um Registro auxiliar no quicksort binário (custo O(1)).

3.4 Mergesort

Por fim, temos o Mergesort, um algoritmo de ordenação que, nesse contexto, realizará divisões recursivas do array de n registros em dois sub-arrays de tamanho n/2 seguido do procedimento de junção ordenada desses sub-arrays (Pos-Ordem). Nessa lógica, obtemos uma Função de Complexidade $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$, que, pelo Teorema Mestre, configura uma complexidade de tempo de O($n \cdot log(n)$). Já a complexidade de espaço do mergesort consiste no custo da entrada de n Registros O(n) e o custo de alocação de memória adicional dos sub-arrays de Registros criados no procedimento merge, o que resulta em uma complexidade de espaço total de O(n) + O(n) = O(n).

3.5 Imprime

O procedimento de impressão tem complexidade de tempo e espaço de O(n), devido ao fato de que o procedimento percorre todas as n linhas uma vez para imprimi-las e não aloca espaço extra para tal.

4 Configuração Experimental

Para a ordenação dos nomes e dados conforme requerido, adotamos 4 diferentes configurações de algoritmos de ordenação dentre os apresentados acima. É importante ressaltar que a ordenação está como requerido: um algoritmo para ordenar o array de Registros em relação ao campo "dado" (ordem crescente) e outro algoritmo para ordena-lo em relação ao campo "nome" (ordem alfabética), nessa ordem.

Em seguida, é impresso na tela o array após a ordenação e calculado o tempo de execução (em milissegundos, aproximado por valores máximos, mínimos e tendências), por meio de repetidos experimentos estritamente no trecho onde é realizada cada uma das configurações de algoritmos. Com isso, o objetivo dos experimentos descritos é avaliar o resultado da ordenação, com o fim de verificar, para cada configuração de algoritmos, se os arrays de Registros estarão ordenados adequadamente e se tempo de execução destes é apropriado. Nesse sentido, temos as seguintes configurações:

Configuração 1: Heapsort e Quicksort Heapsort ordena em relação ao campo "dados". Quicksort ordena em relação ao campo "nome".

Configuração 2: Radix Exchange Sort e Quicksort Radix Exchange Sort ordena em relação ao campo "dados". Quicksort ordena em relação ao campo "nome".

Configuração 3: Heapsort e Mergesort Heapsort ordena em relação ao campo "dados". Mergesort ordena em relação ao campo "nome".

Configuração 4: Radix Exchange Sort e Mergesort Radix Exchange Sort ordena em relação ao campo "dados". Mergesort ordena em relação ao campo "nome".

5 Resultados

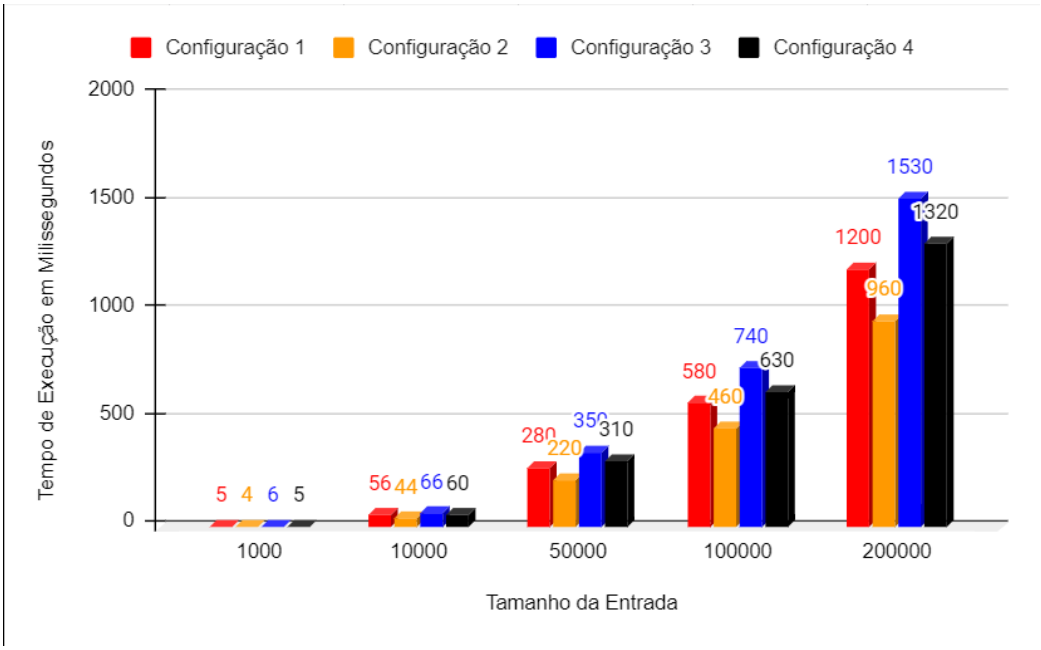

Gráfico com os resultados de tempo de execução por configuração e tamanho da entrada

5.1 Configuração 1:

Ao final do experimento com essa configuração, observou-se um array de registros ordenado em ordem alfabética. Contudo, para uma sequência de nomes iguais, os dados não estão em ordem crescente, o que é explicado pelo fato de que o algoritmo que ordena o array em relação ao campo "nome" é um algoritmo **instável** (nesse caso, o Quicksort), isto é, não preserva a ordem relativa original (criada pelo Heapsort) durante a ordenação de elementos iguais. Possui rapidez intermediária dentre as configurações.

Tabela de registros de tempo da configuração 1 (em ms).

Linhas	Tempo Mínimo	Tempo Máximo	Tempo Real Aproximado
1000	3	6	5
10000	42	65	56
50000	200	380	280
100000	420	780	580
200000	1000 (1s)	1350	1200

5.2 Configuração 2:

Para esta configuração, observamos comportamento semelhante à anterior, isto é, obtemos registros em ordem alfabética, porém os dados não estão em ordem crescente para uma sequência de nomes iguais. Nesse sentido, a desordem do campo dados se deve, novamente, à instabilidade do Quicksort aplicado ao array em relação aos nomes, contudo, agora com o uso do Radix Exchange Sort, a ordenação ocorre em um tempo mais curto. Inclusive, é a configuração mais rápida das quatro analisadas para todos os tamanhos de entrada, que pode ser verificado pelo uso do quicksort normal e a versão binária presente no Radix Exchange Sort, um método semelhante ao quicksort original, inclusive no tempo de execução rápido.

Tabela de registros de tempo da configuração 2 (em ms).

Linhas	Tempo Mínimo	Tempo Máximo	Tempo Real Aproximado
1000	2	5	4
10000	34	52	44
50000	170	300	220
100000	320	510	460
200000	910	1200	960

5.3 Configuração 3:

Usando o Heapsort e o Mergesort obtemos, ao final da ordenação, um array de registros em ordem alfabética e com os dados em ordem crescente para nomes iguais como requisitado, uma vez que o Mergesort é um algoritmo **estável**, já que elementos de igual valor não são trocados de ordem. No entanto, sobre o tempo de execução, essa combinação é a mais lenta de todas, apesar da complexidade de tempo de ambos serem relativamente estáveis com o crescimento do tamanho das entradas.

Tabela de registros de tempo da configuração 3 (em ms).

Linhas	Tempo Mínimo	Tempo Máximo	Tempo Real Aproximado
1000	4	7	6
10000	57	78	66
50000	300	410	350
100000	700	820	740
200000	1250	1700	1530

5.4 Configuração 4:

A última configuração, semelhante à anterior, resulta em um array de Registros ordenado conforme requisitado (nomes em ordem alfabética e dados em ordem crescente), dada a estabilidade do Mergesort. No entanto, os tempos de execução para todas as entradas são levemente mais rápidos que a configuração anterior, o que é explicado pelo quicksort binário presente no Radix Exchange Sort.

Tabela de registros de tempo da configuração 4 (em ms).

Linhas	Tempo Mínimo	Tempo Máximo	Tempo Real Aproximado
1000	3	6	5
10000	51	67	60
50000	285	340	310
100000	590	670	630
200000	1190	1450	1320

6 Conclusões

Com este Trabalho prático, testamos e avaliamos a estabilidade e rapidez de 4 diferentes configurações de algoritmos para a ordenação de registros formatados em nomes e dados binários para cada nome, de modo que a ordenação esperada é: nomes em ordem alfabética e dados em ordem crescente caso haja mais de um dado por nome.

Nesse contexto, os resultados mostrados denotam um ranking de rapidez dessas configurações para todos os tamanhos de entrada: em primeiro, a mais rápida, a configuração 2, em segundo, a configuração 1, em terceiro, a configuração 4 e, em último, a mais lenta, a configuração 3. Contudo, apesar de serem mais rápidos, as configurações 1 e 2 apresentam instabilidades na ordenação dos dados binários, ao passo que as configurações 3 e 4 ordenam os registros como requisitado.

Portanto, mediante observação de velocidade e estabilidade, a configuração 4 (Radix Exchange Sort e Mergesort) se mostrou a mais adequada para resolver o problema descrito adequadamente, pois apresenta o melhor equilíbrio entre a entrega de um array de Registros ordenado conforme pedido e um tempo de execução razoável, ao passo que utiliza algoritmos estáveis e relativamente velozes em comparação com outras configurações.

7 Bibliografia

References

[1] Luiz Chaimowicz e Raquel Prates. *Universidade Federal de Minas Gerais*. [*Material fornecido pela Disciplina de Estrutura de Dados*]. Belo Horizonte, Brasil.

[2] Website público para biblioteca usada para medição dos tempos de execução: *GeeksForGeeks*.
`https://www.geeksforgeeks.org/measure-execution-time-function-cpp/`

8 Instruções para Compilação e Execução

Para compilar o programa, execute o comando 'make' no terminal na pasta raiz do TP. Em seguida, para executar o 'run.out' gerado na pasta 'bin', vá para este diretório executando o comando 'cd bin/'. Por fim, rode o programa com o comando: ./run.out <arquivo de texto com as entradas (.txt)> <configuração a ser utilizada: 1, 2, 3 ou 4 (compatíveis com a numeração descrita nesta documentação)> <Número de linhas a serem ordenadas>.

Exemplos:

1 : ./run.out homologacao.txt 1 200000 (configuração 1 com 200000 linhas do arquivo homologacao);

2 : ./run.out homologacao.txt 3 50000 (configuração 3 com 50000 linhas do arquivo homologacao).

Como requerido, o arquivo homologação.txt não estará presente na pasta "bin" para execução, ou seja, antes de executar as instruções acima, inclua o arquivo com as entradas padronizadas na pasta 'bin'. Note que, caso o comando 'make clean' seja rodado, apenas o executável será apagado, e não o arquivo .txt.