Trabalho Prático 1 - Biblioteca do Filipe

Murilo Vale Ferreira Menezes - 2013030996

Outubro de 2016

1 Introdução

Neste trabalho prático, foi criado um sistema de biblioteca para auxiliar Filipe, um bibliotecário do ICEx, em seu trabalho. Este sistema, baseado em arquivos, funciona da seguinte maneira: recebe uma lista de N livros, com os nomes e a situação (disponível ou emprestado) de cada um. Em seguida, separa os livros por ordem alfabética em um número E de estantes, com L livros em cada. Por fim, o programa deve receber K consultas, retornando a situação de cada livro, estando este disponível, emprestado ou inexistente no acervo. Caso esteja disponível, a consulta deve retornar a posição do livro em sua respectiva estante. Os valores N, E, L e K são recebidos previamente.

Cada livro terá relacionado a si dois valores:

- uma string representando o nome, que consiste apenas nas 26 letras do alfabeto latino, minúsculas (a-z) e de *underscores*.
- um caracter, sendo '0' ou '1', representando o estado do livro. '0' significa que o livro, apesar de existente, está emprestado, enquanto '1' significa que o livro existe e está disponível para empréstimo.

A ordenação dos livros será feita com QuickSort Externo, logo após se receber a primeira lista. O tamanho da área na memória principal será definido de acordo com o limite de livros presentes na memória interna, M, definido pelo usuário. Dado que deve-se ter um livro para guardar temporariamente cada livro do arquivo, a área terá um tamanho de M-1. Após a ordenação, a memória alocada para a área será liberada. Os livros ordenados ficarão em um arquivo denominado **livros_ordenados**.

As estantes serão representadas por E arquivos, preenchidos sequencialmente, até que se chegue no número máximo de livros por estante. Desta forma, algumas estantes poderão ficar vazias. Teremos também um arquivo de texto denominado **indice**, que terá E linhas, cada uma com o primeiro e último livros de cada estante. A pesquisa se iniciará por este arquivo, para se determinar

em qual estante se encontra o livro procurado pelo usuário. Encontrada a estante, será executada uma busca binária no arquivo que representa a estante em questão.

As funções de ordenação e busca nos arquivos serão executadas com a ajuda das funções **fseek()**, **fwrite()** e **fread()**, que movem o ponteiro de acesso ao arquivo, escrevem um registro na posição atual e leem um registro na posição atual, respectivamente.

2 Descrição da solução

2.1 Entrada de parâmetros e livros

O programa começa com a entrada de 5 valores pelo usuário no teclado: N, M, E, L e K:

- N: Número de livros a serem incluídos na biblioteca. Cada livro consiste nos campos **nome** e **estado**, descritos ateriormente.
- M: Número máximo de livros presentes na memória principal. Este número é importante ao se definir o tamanho da área de ordenação ao se ordenar o arquivo com os livros inseridos.
- E: Número de estantes. Cada estante será representada por um arquivo diferente, de nome "estanteX", em que X é o número da estante, estando entre 0 e E-1. Mesmo que não se tenha livros o suficiente para se preencher todas as estantes, serão criados E arquivos, podendo alguns deles ficar vazios.
- L: Número máximo de livros em cada estante. As estantes serão preenchidas até se atingir este número. Desta forma, se tivermos 3 estantes e 8 livros, com L = 5, teremos a estante0 com 5 livros, a estante1 com 3 livros e a estante2 vazia.
- K: Número de consultas a serem feitas, uma vez definidas as estantes.

Após se receber estes valores, o programa recebe os nomes e estados dos N livros a serem adicionados à biblioteca, sendo todos colocados em um mesmo arquivo binário, na ordem de entrada.

2.2 Ordenação

Recebidos os N livros, estes serão ordenados com um QuickSort Externo. Este algoritmo de ordenação se baseia em uma área na memória interna, com M-1 livros, e em 4 ponteiros: 2 para leitura e 2 para escrita, começando no início e no fim do arquivo. Os livros são lidos alternadamente, ora do início, ora do fim do arquivo, movendo-se os ponteiros de leitura para o meio do arquivo a cada livro lido.

Primeiramente, os livros são lidos na área de ordenação. Na área, os livros serão mantidos sempre ordenados, com o auxílio de um QuickSort interno, implementado com a função **qsort()**. Ao se encher a área, os livros lidos serão comparados com a área. Caso a chave seja menor que o menor da área, ou seja, caso venha antes na ordem alfabética, o livro é escrito de acordo com o ponteiro de escrita no início do arquivo. Caso seja maior que o maior da área, é escrito de acordo com o ponteiro de escrita no fim do arquivo. Garante-se que os ponteiros de leitura estejam sempre à frente dos ponteiros de escrita, para não ocorrer de escrever-se em uma posição ainda não lida. Caso o livro lido estiver entre o maior o menor livros da área, será feita uma escolha de qual livro será retirado da área: o maior ou o menor, sendo este escrito no arquivo e dando lugar ao novo livro. A área então é ordenada novamente com o QuickSort. A escolha é dada de acordo com o tamanho das áreas já escritas, de modo a se fazer um arquivo mais balanceado o possível.

Ao fim, a área é escrita no arquivo, de modo que todos os livros presentes antes desta área tenham chaves menores, e todos os livros depois da área tenham chaves maiores. O algoritmo é então executado recursivamente em cada uma destas duas regiões. O caso base da recursão se dá quando existe apenas um livro a ser ordenado.

Teremos então um arquivo, **livros_ordenados**, contendo todos os livros, dispostos por ordem alfabética de seus nomes.

2.3 Organização das estantes

Passada a ordenação, são criados E arquivos de nome "estante0"a "estante(E-1)", onde serão organizados os livros. Os nomes dos arquivos são criados com ajuda da função **strcat()**, que concatena a string "estante"com o número relacionado. As estantes serão salvas em um array de ponteiros para arquivo, **est**.

Criados os arquivos, os livros são lidos sequencialmente no arquivo **livros_ordenados**, sendo delegados aos arquivos, começando pelo "estante0". Atingidos L livros, o arquivo de destino é iterado. Paralelamente, é escrito o arquivo de texto **indice** sempre que tivermos o primeiro ou o último livros de cada estante. Este processo continua até que se leia todos os livros de **livros_ordenados**.

2.4 Busca de livros

A busca é separada em duas etapas, executadas em funções distintas:

- buscaEstante(): esta função recebe o nome do livro a ser procurado e o arquivo de índices e retorna o número da estante correspondente, se encontrado. Caso contrário, retorna um valor que indica que o livro não existe na biblioteca.

- buscaLivro(): esta função recebe o arquivo da estante correspondente no array de arquivos, o nome do livro buscado e ponteiros para o início e fim do arquivo. Retorna a posição do livro, caso disponível. Caso o livro exista, mas esteja emprestado (estado == 0), retorna -1. Caso o livro não exista, retorna -2.

Assim, na primeira função é lida cada linha do arquivo até encontrar-se a estante correspondente ao livro buscado, retornando-se seu número. Caso a chave do livro esteja, por exemplo, antes da chave do primeiro livro, depois do último ou entre duas estantes, será retornado o valor -1, indicando que o livro não se encontra em nenhuma estante.

Caso se encontre a estante, é executada a segunda função, que executa uma busca binária no arquivo da estante de índice descoberto anteriormente. Então, de acordo com o valor retornado pela função, saberemos a situação do livro e sua posição na estante, caso disponível.

3 Análise de complexidade

A seguir será feita uma análise da complexidade de cada função implementada.

3.1 Temporal

- stringEstante(): esta função recebe um número X e retorna uma string "estanteX". Tem tempo de execução constante para qualquer entrada. Logo, complexidade O(1).

- buscaEstante(): esta função percorre o arquivo de índices à procura da estante correspondente ao livro buscado. No pior caso, esta função percorre todas as E linhas, com exatamente 2 nomes em cada linha. Em cada linha, as operações de leitura têm complexidade constante. Assim, a complexidade é O(E).

- buscaLivro(): esta função exerce uma busca binária no arquivo da estante correspondente. Cada arquivo possui no máximo L livros. Considera-se a complexidade das operações de leitura e de *seek* como O(1). Desta forma, a complexidade de buscaLivro() é $O(\log_2 L)$.

- comparaChaves(): esta função, implementada para auxiliar o QuickSort interno, recebe dois livros e compara as chaves, ou seja, os nomes de acordo com a ordem alfabética. Utiliza-se da função strcmp(). O tempo de execução desta função é constante, sendo sua complexidade O(1).

- qsortExt(): de acordo com Ziviani [1], no melhor caso, como em um arquivo previamente ordenado, a complexidade do QuickSort Externo é

O(N). O pior caso acontece quando uma das partições de antes ou depois da área ordenação fica vazia, enquanto a outra tem o tamanho máximo. A complexidade neste caso é $\mathbf{O(N^2/TamArea)} = \mathbf{O(N^2/(M-1))}$. No caso médio, a complexidade é $\mathbf{O(N * log(N/(M-1)))}$.

- main(): ao ler-se os livros a serem gravados, serão executadas operações de leitura com complexidade constante, para cada livro. Então, a complexidade desta etapa é O(N). Ao se ordenar o arquivo com os livros, espera-se a complexidade do caso médio do QuickSort Externo, $\mathbf{O(N * log2(N/(M-1)))}$. Ao criar-se os arquivos com as estantes, temos a complexidade temporal de O(E). Ao se escrever os livros do arquivo **livros_ordenados** aos arquivos das estantes, temos a complexidade de O(N), uma vez que são executadas operações O(1) para cada livro. Por fim, na busca, contando as duas etapas, teremos a complexidade de $O(E + \log_2 L)$, para cada consulta. Assim, podemos esperar uma complexidade total do programa como $\mathbf{O(N + N * log2(N/(M-1)) + K*E + K*log_2 L)}$.

3.2 Espacial

A complexidade espacial deste programa é relativamente baixa, uma vez que a grande maioria dos dados fica gravado em memória secundária. Serão utilizados M livros na memória interna e um array de E ponteiros para arquivos. As outras variáveis alocam a mesma quantidade de memória, independentemente dos parâmetros. Assim, a complexidade espacial é O(M + E).

4 Análise experimental

4.1 Espacial

Para analisar-se a complexidade espacial, foi usada a função mallinfo(), biblioteca malloc.h, que retorna o número de bytes alocados em determinado momento do programa. Os resultados foram obtidos antes de se liberar toda a memória alocada dinamicamente. Anteriormente, concluiu-se que a complexidade era O(M + E). Desta forma, variou-se as duas entradas independentemente, provando-se que o crescimento é linear.

Variando-se M, foram utilizados os limites de livros na memória interna de 2, 3, 4, 5, 10, 20, 50 e 100 livros. O resultado é dado pelo gráfico a seguir.

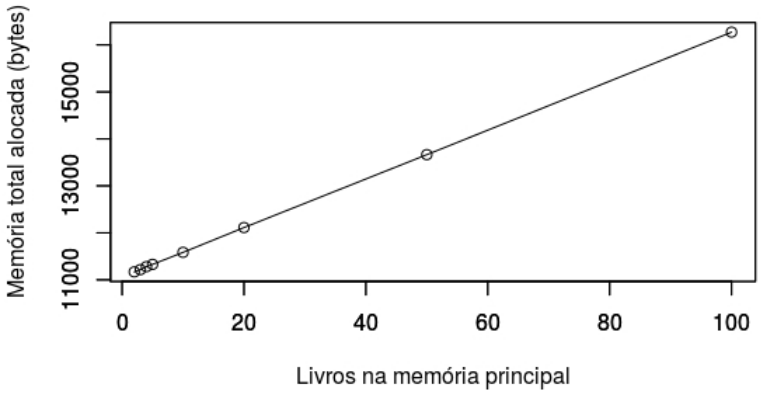

Figura 1: Gráfico gerado pela análise da complexidade espacial, variando-se o valor de M

Variando-se E, foram utilizadas 1, 2, 3, 4, 5, 10, 20, 50 e 100 estantes. O resultado é dado pelo gráfico a seguir.

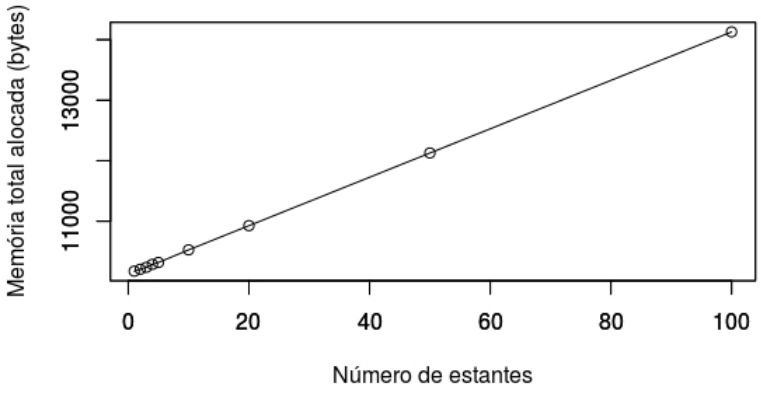

Figura 2: Gráfico gerado pela análise da complexidade espacial, variando-se o valor de E

4.2 Temporal

Para se analisar o crescimento do tempo de execução, foi usada a função clock(), da biblioteca time.h, que fornece a quantidade de ciclos de clock executados desde o início da execução do programa. Para cada teste, foram executados 30 vezes o programa com as mesmas entradas, tirando-se a média.

Vemos que o tempo de execução do programa cresce, na média, assintoticamente em $\mathbf{O(N + N * log2(N/(M-1)) + K*E + K*log_2 L)}$. Desta forma, ao variar-se cada uma das variáveis, devemos ter respostas da seguinte maneira:

- N: Com o aumento de N, o tempo de execução deve crescer com N*logN.
- M: Com o aumento de M, o tempo de execução deve decrescer com o logaritmo do inverso de M.
- E: Com o aumento de E, o tempo de execução deve crescer linearmente.
- L: Com o aumento de L, o tempo de execução deveria crescer de forma logaritmica. Porém, não se pode confiar na análise experimental da função main() para este parâmetro, já que um aumento de L acarreta em uma diminuição do número de estantes que efetivamente têm livros, o que descaracteriza a curva dependente apenas de L, como mostra o gráfico abaixo.

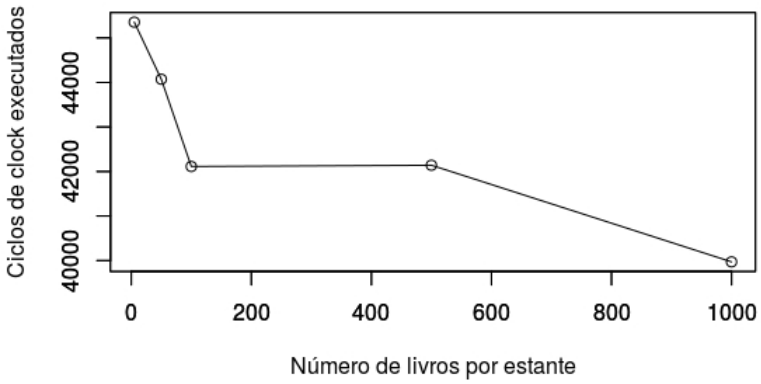

Figura 3: Gráfico gerado pela análise da complexidade temporal, variando-se apenas o valor de L na função main

- K: Com o aumento de K, o tempo de execução deve crescer linearmente.

Fixando-se as outras variáveis e aumentando N para valores de 5, 10, 20, 50, 100, 150, 200, 500, 800 e 1000, obteve-se o seguinte gráfico:

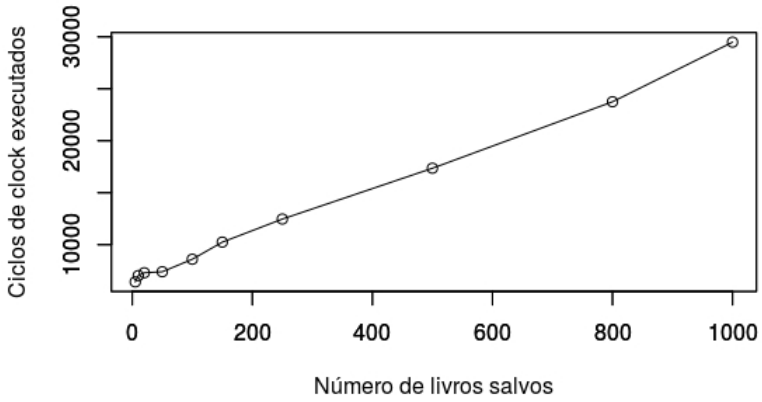

Figura 4: Gráfico gerado pela análise da complexidade temporal, variando-se o valor de N

Variando-se o valor de M para 5, 10, 20, 40, 80, 150 e 250, obteve-se o seguinte gráfico:

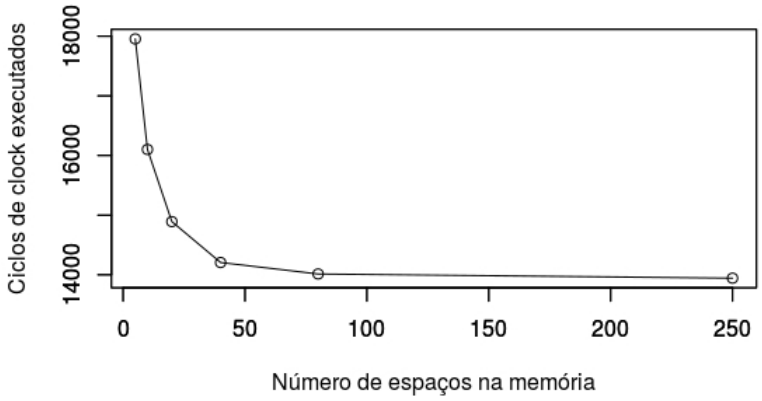

Figura 5: Gráfico gerado pela análise da complexidade temporal, variando-se o valor de M

Variando-se o valor de E em 1, 2, 5, 10, 20, 50 e 100 estantes:

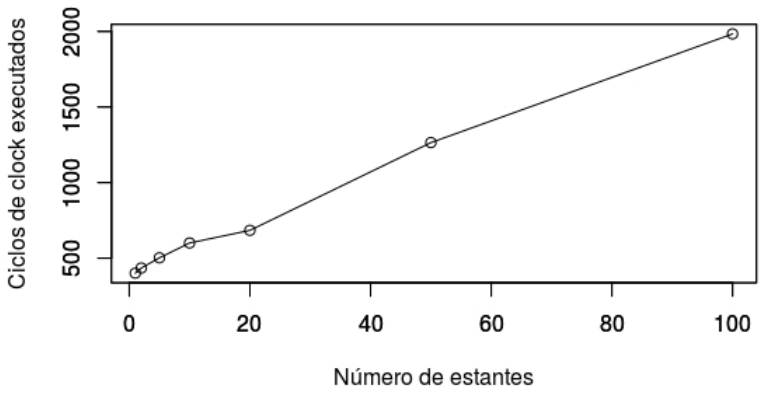

Figura 6: Gráfico gerado pela análise da complexidade temporal, variando-se o valor de E

Variando-se o valor de K em 5, 25, 50, 100, 250, 500 e 1000:

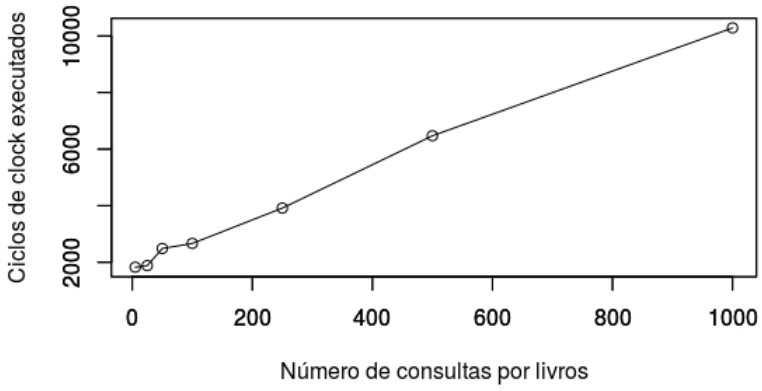

Figura 7: Gráfico gerado pela análise da complexidade temporal, variando-se o valor de K

5 Conclusão

Pode-se concluir que as análises de tempo e de espaço foram previsíveis, de acordo com as deduções feitas anteriormente. No caso do parâmetro L, não pode se verificar a dedução pois este não age independentemente dos outros, com sua variação mudando o comportamento de outras partes do programa.

Os algoritmos envolvidos foram de implementação relativamente simples, assim como a manipulação de arquivos.

E, acima de tudo, podemos concluir que Filipe poderá ser muito mais produtivo em seu trabalho.

Referências

[1] Nivio Ziviani et al. *Projeto de algoritmos: com implementações em Pascal e C*, volume 2. Thomson, 2004.