Introdução à Análise de Algoritmos
2º semestre de 2013 - Turma 94

Segundo Exercício Programa/Trabalho

Busca e Ordenação

1 Introdução

A proposta deste exercício programa é exercitar os conceitos discutidos em aula sobre busca e ordenação. Neste EP tanto busca quanto ordenação serão realizados sobre um conjunto de strings, representando um conjunto de palavras. Vocês devem implementar duas classes executáveis: **Ordenador** e **Buscador**. A classe **Ordenador** deve implementar os seguintes algoritmos de ordenação: *selection sort* e *heapsort*. Já a classe **Buscador** deverá implementar a busca sequencial, busca binária e busca através de tabelas de espalhamento.

Além de implementar estes algoritmos de ordenação e busca, vocês deverão realizar alguns experimentos e produzir um relatório com os resultados obtidos. Tanto para os algoritmos de ordenação, quanto para os algoritmos de busca, as analises dos resultados deverão levar em conta os dois critérios distintos, embora diretamente relacionados entre si: número de comparações realizadas para a execução de um algoritmo e tempo que um algoritmo leva para ser executado. Maiores detalhes sobre cada uma das classes que devem ser implementadas, formatação de entrada e saída, assim como os experimentos a serem executados são detalhados nas seções a seguir.

2 Ordenação

A execução da classe **Ordenador** deve se feita da seguinte forma:

```
java Ordenador config.txt
```

onde `config.txt` é um arquivo de configuração, em formato texto, que possui a seguinte estrutura:

```
<algoritmo>
<debug>
<arquivo de palavras>
<arquivo ordenado>
```

O parâmetro `algoritmo` define qual o algoritmo de ordenação utilizado, devendo assumir o valor **selection**, para escolher o *selection sort*, ou **heap**, para escolher o *heapsort*. O parâmetro `debug` deve assumir valor **true** ou **false** e deve ser usado para habilitar ou não o modo de depuração. Já o terceiro parâmetro determina o nome do arquivo que contem a lista de palavras a serem ordenadas. Este arquivo também deve estar em formato texto e sua primeira linha deve conter a quantidade n de palavras presentes no arquivo. Seguindo esta primeira linha, devem haver mais n linhas no arquivo, cada uma com uma palavra. Finalmente, o último parâmetro define o nome do arquivo de saída, que deverá conter as palavras ordenadas em ordem lexicográfica crescente (também com uma palavra por linha).

Quando o modo de depuração estiver habilitado, a classe **Ordenador** deve imprimir na saída padrão algumas informações adicionais, de modo se a verificar passo a passo o andamento da ordenação. As informações adicionais que devem ser impressas são:

1. Estado inicial do vetor.
2. Estado do vetor após ter sido transformado em um heap (apenas para o algoritmo *heapsort*).
3. Estado do vetor ao final de cada execução do laço principal da ordenação (observe que o estado a ser impresso ao final da última execução do laço principal deve corresponder ao vetor ordenado).
4. Número de comparações realizadas e tempo que a ordenação levou para executar.

Note que imprimir o "estado do vetor" corresponde a imprimir todos os seus elementos, em uma mesma linha, separando-os por espaço. Assim, a saída gerada pela classe **Ordenador**, quando o modo depuração estiver habilitado, será composta por várias linhas, e cada linha corresponderá a uma permutação dos elementos do vetor (com exceção da última linha que informará o número de comparações e o tempo que a ordenação levou para ser feita).

3 Busca

A classe **Buscador** deve ser executada de modo similar à classe **Ordenador**:

```
java Buscador config.txt
```

onde o arquivo de configuração `config.txt` possui a seguinte estrutura (similar ao arquivo de configuração do **Ordenador**, mas com alguns parâmetros a mais):

```
<algoritmo>
<arquivo de palavras>
<arquivo de saída>
<numero de buscas>
<palavra 0>
<palavra 1>
<palavra 2>
...
<palavra k-1>
```

O parâmetro `algoritmo` deve ser especificado com um dos seguintes valores: **seq** (para escolher pela busca sequencial), **bin** (para escolher pela busca binária), **hash1** (para busca usando tabela de espalhamento) ou **hash2** (também para busca usando tabela de espalhamento, mas usando uma função de *hashing* diferente — detalhes sobre como deve funcionar a tabela de espalhamento, incluindo as funções de *hashing* serão detalhadas adiante). O segundo parâmetro define o nome do arquivo que contem a lista de palavras sobre as quais as buscas serão realizadas (primeira linha contem a quantidade n de palavras, seguida por n linhas, cada uma com uma palavra). O terceiro parâmetro define o nome do arquivo de saída onde os resultados das buscas serão registrados. O parâmetro seguinte determina o número k de buscas que serão realizadas. Seguindo esta linha haverá uma sequência de k linhas, cada uma contendo uma palavra a ser pesquisada. O arquivo de saída para as buscas deve apresentar a seguinte formatação:

```
<palavra 0> <resultado> <número de comparações> <tempo de busca>
<palavra 1> <resultado> <número de comparações> <tempo de busca>
...
<palavra k-1> <resultado> <número de comparações> <tempo de busca>
```

Cada uma das k linhas do arquivo de saída apresentam informações referentes às buscas de cada palavra definida no arquivo de configuração. Nestas linhas: `resultado` deve ser a string **SIM** (para uma busca bem sucedida) ou a string **NAO** (quando a palavra pesquisada não é encontrada — note que a string **NAO** não deve estar acentuada para evitar eventuais problemas devido a *encodings* diferentes em plataformas distintas); `número de comparações` é um valor inteiro que indica a quantidade de comparações que foram realizadas na busca; e `tempo de busca` é um valor inteiro que indica quantos milissegundos a busca da palavra demorou para ser realizada.

Note que o `tempo de busca` de uma palavra deve contabilizar única e exclusivamente o tempo que a busca leva para ser feita **para a palavra em questão**, descartando o tempo gasto para carregar/preparar os dados. Observe ainda que, a fim de se fazer comparações precisas entre os tempos de busca para diferentes algoritmos, deve-se manter o conjunto de palavras sobre as quais a busca será feita totalmente carregado em memória. Para cada um dos algoritmos de busca isso significa:

- **Busca sequencial**: o conjunto de palavras deve ser mantido em memória como um vetor de strings. Neste caso, o tempo de carga/preparo corresponde ao tempo gasto para fazer a leitura do arquivo de palavras e preencher o vetor de strings.
- **Busca binária:** o conjunto de palavras deve ser mantido em memória como um vetor **ordenado** de strings. Neste caso, o tempo de carga/preparo corresponde ao tempo gasto para fazer a leitura do arquivo de palavras e preencher o vetor de strings, mais o tempo necessário para ordenar o vetor. Use um dos algoritmos de ordenação já implementados na classe **Ordenador**, de preferência o mais eficiente deles!
- **Busca por tabela de espalhamento**: o conjunto de palavras deve ser mantido em memória como uma tabela de espalhamento. Neste caso o tempo de carga/preparo envolve o tempo gasto para se fazer a leitura do arquivo de palavras e realizar as inserções na tabela.

Note que a carga/preparo é um processo que é executado apenas uma vez, independente quantas buscas são realizadas em seguida, e que o tempo gasto com carga/preparo **não** deve ser levado em conta ao se contabilizar o tempo de busca de uma palavra.

3.1 Sobre a implementação da tabela de espalhamento (*hash tables*)

A tabela de espalhamento a ser implementada deve possuir $n/10$ *slots*, onde n é o total de palavras que a tabela irá armazenar, e colisões devem ser resolvidas através de encadeamento. Em relação às funções de *hashing*, devem ser implementadas duas funções distintas, lembrando que a função utilizada na execução do EP deverá ser escolhida através do arquivo de configuração. Quando o algoritmo escolhido for **hash1**, a função de *hashing* para uma dada palavra (ou seja, uma string) p deve ser calculada segundo a fórmula abaixo:

$$h(p) = (p[0] \times 2^{16} + p[1] \times 2^8 + p[2]) \,\%\, s \tag{1}$$

onde: $p[0]$, $p[1]$ e $p[2]$ correspondem, respectivamente, ao valor numérico do primeiro, segundo e terceiro caracteres presentes na palavra p; e s corresponde ao número de *slots* da tabela, ou seja, $s = n/10$. No caso de palavras que possuem menos de 3 caracteres, deve-se assumir o valor zero para os caracteres inexistentes.

Vocês são livres para implementar a segunda função de *hashing* (a ser utilizada na execução do EP quando o algoritmo escolhido for **hash2**) da maneira que quiserem, mas ela deve ser uma função "melhor" do que a especificada na Equação 1. Não se esqueçam de documentar no relatório como essa segunda função de *hashing* é calculada, além de justificar a escolha pela mesma.

4 Experimentos e relatório

Além da implementação dos algoritmos de ordenação e busca, vocês também devem realizar alguns experimentos práticos e redigir um relatório a partir dos resultados obtidos. Um roteiro mais detalhado sobre como os experimentos deverão ser conduzidos será disponibilizado em breve.

5 Prazos e entrega

A entrega do EP deve ser feita até o dia 18/01/2014 às 23:55 pelo TIDIA-Ae. Entregue um arquivo zip contendo: os arquivos fontes do EP; um README explicando como compilar e rodar seu EP; e o relatório em formato PDF. Este EP pode ser feito em grupos de até 3 pessoas.

Boa diversão!