Universidade de Évora

Disciplina de Estruturas de Dados e Algoritmos I

Corrector Ortográfico

Autores:
Ricardo Fusco, 29263
Luís Candeias, 29615

Professor:
Lígia Ferreira

28/01/2014

Índice

Lista de Figuras

1 Introdução

No âmbito da unidade curricular de Estruturas de Dados I pretende-se, através das estrutura abordadas nas aulas, nomeadamente as tabelas de Hash, criar um corrector ortográfico simples que gera como output 4 ficheiros, que irão conter as palavras erradas e respectivas linhas, as palavras erradas e respectivas sugestões por inserção de uma letra, remoção de uma letra e troca de duas letras adjacentes. Para tal é necessário carregar um dicionário, fornecido pela professora, para uma hashtable tendo em conta os diferentes tipos de acesso (Linear, quadrático e duplo), e após carregar o dicionário ter-se-à que ler um ficheiro de texto e verificar as palavras erradas para posterior análise de sugestões a escrever nos 4 ficheiros gerados como output. Para correr o programa basta executar o método gerarOutput() que chama as funções para escrever os ficheiros de output pretendidos.

2 Desenvolvimento

Para realizar este trabalho foram usadas diferentes estruturas acerca das quais iremos falar mais a frente. Essas estruturas são as HashTables, ArrayLists e Tuplos. A implementação de ArrayLists foi criada como auxiliar para armazenar as palavras erradas e respectivas linhas, ou as palavras erradas e respectivas sugestões. A implentação de Tuplos foi feita para que possamos armazenar o par (palavra errada, linha) ou (palavra errada, sugestoes) para facilitar um pouco a leitura e facilitar também a escrita para o ficheiro tornando tudo mais organizado. No que diz respeito às HashTables, estas eram fulcrais para carregar o dicionário que vai servir de base ao corrector ortográfico.

2.1 Estruturas Utilizadas

2.1.1 Hashtables

A implementação de HashTables utilizada para o trabalho é a mesma que elaboramos para a ultima aula pratica da cadeira com algumas modificações. Temos uma classe ***ElementoTabela*** $< T >$ com 2 variáveis, uma para o "elemento" do tipo T e outra variavel booleana "activo" para controlar se o elemento está activo ou não, e por activo quer-se dizer, se o elemento ja foi apagado ou não da hashtable. Esta classe contém apenas os construtores para o instanciamento de um objecto do tipo ElementoTabela, e o método toString() que devolve uma representação em String dum elemento de uma tabela.

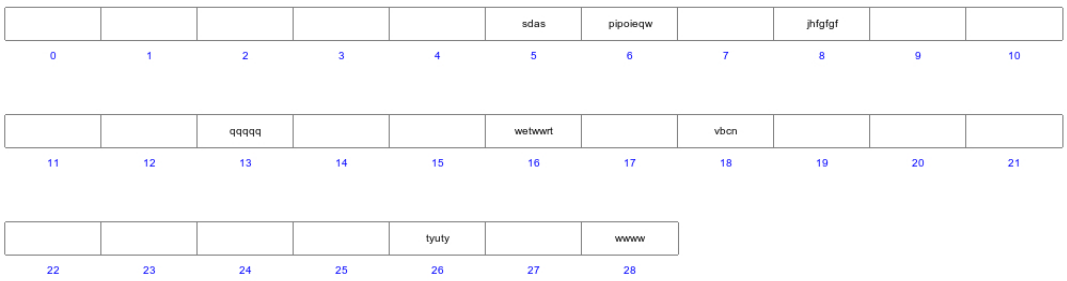

Figure 1: Exemplo Tabela Hash (Strings)

No que diz respeito à tabela em sí temos uma classe abstrata ***HashTable*** $< T >$ com duas variáveis, uma variável "ocupados" que serve como controlo do número de posições ocupadas na hashtable, e um array tabela que contém objectos do tipo ElementoTabela. Esta classe implementa os seguintes métodos:

- ***private boolean estaActivo*** **(*int currentPos*)** - este método devolve true se o elemento na posição que recebe como argumento estiver activo e não for "null".
- ***public int ocupados*** **()** - este método devolve o número de posições ocupadas na HashTable.

- ***protected int hash (T s)*** - este é um método que calcula o hashing de um objecto que recebe como parâmetro. Esta função de hash em particular está optimizada para Strings utilizando a regra de Horner incluindo todos os caracteres da String resultando numa distribuição razoavelmente bem distribuída fazendo desta uma boa função de cálculo de hash de strings. Caso o resultado seja maior que o tamanho da tabela de hash é calculado o resto da divisão do resultado pelo tamanho da tabela.

- ***public int capacidade*** () - este método devolve o tamanho da tabela

- ***public void alocarTabela (int dim)*** - cria uma nova tabela de ElementosTabela de tamanho igual ao inteiro que recebe como argumento e aloca esse espaço criado para a array tabela.

- ***private static boolean ePrimo (int n)*** - este método verifica se o inteiro que recebe como argumento é um número primo(true) ou não(false).

- ***private static int proxPrimo (int n)*** - este método encontra e devolve o número primo a seguir do número que recebe como argumento usando o método ePrimo para esse efeito.

- ***public void tornarVazia*** () - este método coloca todas as posições da HashTable a "null" e faz um reset no numero de posiçoes ocupadas da mesma (ocupados = 0).

- ***public boolean procurar (T x)*** - método que devolve true se encontrar na hashtable o objecto que recebe como argumento e false caso contrário.

- ***public boolean remove (T x)*** - remove um objecto da tabela de hash fazendo a pesquisa do elemento atravéz do calculo do hash do mesmo se estiver activo coloca-o inactivo (activo = false), decrementa o numero de posições ocupadas e devolve true, se o elemento já estiver apagado (activo = false) não faz nada, apenas devolve false.

- ***public void insere (T x)*** - insere um elemento na tabela de hash e incrementa o numero de posições ocupadas. E verifica se, após a inserção do elemento, a tabela ficar com mais de metade da sua capacidade ocupada é feito o rehash da tabela.

- ***public void rehash*** () - neste método é duplicado o tamanho da hashtable criando uma nova tabela vazia com o dobro do tamanho que tinha, é feito um reset ao numero de posições ocupadas e os elementos sao copiados da tabela antiga para a nova.

- ***public String toString*** () - por fim temos o método que devolve uma representação da hashtable sob a forma de string. O método print() apenas faz o system.out.println() do toString.

Por fim resta o método abstracto ***protected abstract int procPos (T s)*** que será implementado nas classes ***LinHashTable***, ***QuadHashTable*** e ***DoubleHashTable***. É neste método que irão ser implementados os vários tipos de acesso. As 3 classes anteriormente referidas irão fazer o extend da classe abstrata ***HashTable < T >***

herdando assim todos os métodos dessa classe e tendo obrigatóriamente que implementar o método procPos(T s). No caso da classe com o acesso duplo irá estar implementada uma segunda função de hash simples para este tipo de acesso .

2.1.2 ArrayLists

Além das HashTables foi criada uma implementação de ArrayLists, como estrutura auxiliar para armazenar tuplos com as palavras erradas e as linhas onde ocorrem, e sugestões associadas a essa palavra. As variáveis da classe são o size (tamanho da ArrayList) e a array de objectos de um tipo e a variável tamPorDefeito que irá definir um tamanho por defeito caso crie uma instância de ArrayLists sem argumento.

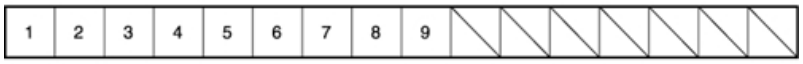

Figure 2: Exemplo ArrayList

Esta classe implementa os seguintes métodos:

- ***public int sizeOf*** () - este método devolve o tamanho da arraylist.
- ***public boolean isEmpty*** () - retorna true se o tamanho da arraylist for zero e false caso contrário.
- ***public T get*** (***int ind***) - este método devolve o elemento da arraylist na posição que recebe como parâmetro.
- ***public boolean contains*** (***T x***) - este método itera sobre os elementos da arraylist e devolve true se encontrar na mesma objecto que recebe como argumento.
- ***public T set*** (***int ind***, ***T novoVal***) - método que troca um objecto na arraylist por um novo objecto recebendo como argumento o índice do objecto a alterar e o objecto a inserir (trocar).
- ***public void updateSize*** (***int newCapacity***) - este método é semelhante ao método rehash() das hashtables, tendo como função a criação de uma nova arraylist de tamanho igual ao que recebe como argumento e população do novo arraylist com os elementos que se encontravam no arraylist antigo.
- ***public boolean insert*** (***T x***) - neste método é chamado o método ***public boolean insert*** (***int ind***, ***T x***) que insere o objecto x numa posição da arraylist, que recebe como argumento,que neste caso é no fim da mesma (ind = size).
- ***public T remove*** (***int ind***) e ***public T remove*** (***T x***) - estes são os métodos existentes para a remoção de um objecto na arraylist seja procurando o objecto por indice ou procurando pelo próprio objecto, dependendo do que receber como argumento(int ind ou T x).
- ***public void makeEmpty*** () - este método faz um reset ao tamanho da arraylist (size = 0) e faz um update da arraylist esvaziando-a.

- ***public String toString*** **()** - método toString desta classe que devolve uma representação da arraylist na forma de string.

Há também a classe ArrayListIterator que implementa os métodos habituais dum iterador como next(), hasNext() e remove(). Esta classe está declarada dentro da classe da ArrayList permitindo que o iterador aceda a variáveis e métodos declarados como privados e ao mesmo tempo permancer privada apenas desta classe.

2.1.3 Tuplos

Foram também utilizados tuplos como estrutura auxiliar para o trabalho. Os tuplos são utilizados, como já foi referido anteriormente, no sentido de guardar as palavras erradas e a linha onde ocorrem, as palavras erradas e a lista de sugestões para qualquer dos casos. A classe Tuplo é apenas constituida por duas variáveis, um x e um y de um qualquer tipo, pelo construtor onde se inicializam as variáveis e o método toString que retorna uma representação string dum tuplo.

2.2 Corrector

Por fim temos a classe Corrector onde implementamos os métodos com os algoritmos para gerar as sugestões e os métodos para escrever as respectivas sugestões para os ficheiros. Temos apenas uma variável, uma variável do tipo hashtable com um número de elementos escolhido de modo a obter uma tabela de hash o mais pequena possível de modo a que seja maximizada a performance e a velocidade do calculo das sugestões. O tipo de acesso utilizado por nós foi o acesso linear. A classe corrector contém os seguintes métodos:

- ***public void popularHashTable*** **(*String s*)** - é neste método que é lido o ficheiro de dicionário, que irá servir como base para o corrector, e à metida que são lidas as palavras do ficheiro, estas são inseridas na hashtable.
- ***public ArrayList*** $<$ ***Tuplo*** $>$ ***palavrasErradas*** **(*String ficheiro*)** - neste método é criada uma arraylist que irá conter todas as palavras que se encontram no texto que pretendemos corrigir, sendo cada posição da arraylist representada por um tuplo $->$ (palavra,linha). De seguida é verificado, para cada palavra na arraylist, se estão na hashtable; caso não estejam é adicionado o tuplo a outra arraylist de tuplos que no fim irá conter apenas as palavras erradas.
- ***public ArrayList*** $<$ ***String*** $>$ ***criaAlfabeto*** **()** - este método cria uma lista com as várias letras do alfabeto (incluíndo caracteres especiais) de acordo com as palavras que se encontram na hashtable. Esta lista irá ser necessária para o algoritmo que obtém as sugestões a partir da inserção de 1 letra.
- ***public ArrayList*** $<$ ***String*** $>$ ***adicionaLetra*** **(*String palavra*)** - este método itera sobre os elementos da arraylist com as letras do alfabeto e percorre cada indice da palavra que recebe como parâmetro inserindo a letra nesse indice e verificando, se o resultado estiver na hashtable é adicionada essa palavra à arraylist das sugestões.

- ***public ArrayList < String > removeLetra (String palavra)*** - este método itera sobre os elementos da palavra que recebe como argumento e a medida que itera remove a letra no indice sobre o qual esta a iterar e procura pela palavra resultante na hashtable, se for encontrada é adicionada a lista das sugestões.

- ***public ArrayList < String > trocaLetras (String palavra)*** - este método itera sobre os elementos da palavra que recebe como argumento convertendo a string para um array de chars e trocando os chars adjancentes (i,i+1) para depois reconverter de volta para string, se o resultado estiver na hashtable é adicionada a palavra resultante à lista das sugestões.

- Os métodos de escrita(***writePalErradas***(), ***writeSugestoesIns***(), ***writeSugestoesRem***() e ***writeSugestoesTroca***()) fazem no fundo a mesma coisa, ou seja, vai ser percorrida a lista com as palavras erradas que recebem como argumento e vai ser chamado o método para dar as sugestões dependendo do caso (remoção de 1 letra, inserção de 1 letra ou troca de 2 letras adjacentes). À medida que a lista das palavras erradas vai sendo iterada vai sendo adicionado o tuplo (palavra, sugestoes) a uma nova lista. Por fim é iterada a lista final e são gravadas as sugestões no ficheiro de acordo com essa ultima arraylist.

- ***public void geraOutput*** () - por fim temos o último método que é aquele onde são chamados todos os outros métodos no sentido de gerar e escrever os 4 ficheiros de output.

Os ficheiros de output ficarão qualquer coisa do género:

sugestoesTroca.txt ×

```
####################################
# Sugestoes por troca de uma letra #
####################################

Palavra errada: Existem
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: vairações
  Sugestoes:
    -variações

Palavra errada: pasagens
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: Lorem
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: Ipsum
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: fomra
  Sugestoes:
    -forma

Palavra errada: innjecção
  Sugestoes:
    - Nao existem sugestoes para esta palavra.
```

Figure 3: Sugestões por troca de 1 letra

sugestoesRemocao.txt ×

```
#########################################
#  Sugestoes por remocao de uma letra   #
#########################################

Palavra errada: Existem
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: vairações
  Sugestoes:
    -varações

Palavra errada: pasagens
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: Lorem
  Sugestoes:
    -orem

Palavra errada: Ipsum
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: fomra
  Sugestoes:
    -fora

Palavra errada: innjecção
  Sugestoes:
    -injecção
```

Figure 4: Sugestões por remoção de 1 letra

sugestoesInsercao.txt ×

```
#########################################
# Sugestoes por insercao de uma letra #
#########################################

Palavra errada: Existem
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: vairações
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: pasagens
  Sugestoes:
    -paisagens
    -passagens
    -passagens
    -pastagens

Palavra errada: Lorem
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: Ipsum
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: fomra
  Sugestoes:
    - Nao existem sugestoes para esta palavra.

Palavra errada: innjecção
  Sugestoes:
    - Nao existem sugestoes para esta palavra.
```

Figure 5: Sugestões por inserção de 1 letra

palavrasErradas.txt ×

```
########################################
# Palavras Erradas e respectivas linhas #
########################################

Palavra Errada: Existem
Este erro ocorre na linha: 1

Palavra Errada: vairações
Este erro ocorre na linha: 1

Palavra Errada: pasagens
Este erro ocorre na linha: 1

Palavra Errada: Lorem
Este erro ocorre na linha: 1

Palavra Errada: Ipsum
Este erro ocorre na linha: 1

Palavra Errada: fomra
Este erro ocorre na linha: 2

Palavra Errada: innjecção
Este erro ocorre na linha: 2

Palavra Errada: palavrs
Este erro ocorre na linha: 3

Palavra Errada: aleatóris
Este erro ocorre na linha: 3

Palavra Errada: praecem
Este erro ocorre na linha: 3
```

Figure 6: Palavras erradas e respectiva linha

3 Conclusão

Em conclusão achamos que conseguímos com sucesso cumprir todos os objectivos deste trabalho. Tivemos alguns problemas inicialmente com os caracteres especiais (com acentos e cedilhas), pois era necessário converter o ficheiro para um formato com codificação UTF-8 pois na codificação em que estava ao colocar as palavras na hashtable, às que tivessem caracteres especiais eram-lhes substítuidos estes caracteres pelo caracter unicode.

Resolvido este problema deparamo-nos com outro, o número de palavras que estavam nas tabela de hash era apróximadamente 50.000 a menos que o numero de palavras no ficheiro de dicionário, inicialmente pensávamos que o problema estaria na nossa implementação até que nos apercebemos do facto de haverem várias palavras repetidas no ficheiro. Os algoritmos para obter as sugestões de uma determinada palavra foram relativamente simples, bastou iterar pela palavra e ir inserindo/removendo/trocando letras e verificar se a palavra resultante dá "hit" na tabela (se está na tabela).

O único aspecto que pode não estar muito optimizado é o algoritmo que verifica as sugestões para uma palavra por inserção de uma letra, visto que este demora um pouco mais do que os outros método. Esta "lentidão" no que diz respeito a este método é devida ao facto de ter de gerar o alfabeto a partir de todas as palavras que se encontram na hashtable multiplas vezes. Este pequeno problema foi resolvido criando o alfabeto antes de chamar este método - adicionaLetra() - para cada palavra errada e passando-o como argumento da mesma, isto veio-nos minimizar o tempo de execução do programa aumentando considerávelmente a performance do mesmo, pois deixou de estar a ser chamado um método desnecessáriamente por cada palavra errada existente, e ainda por cima um método que percorre todos os elementos da hashtable. Contudo, apesar destes pequenos pormenores, conseguimos finalizar este trabalho.