Escola de Ciências e Tecnologia

Departamento de Informática

Licenciatura em Engenharia Informática

Unidade curricular Estrutura e Dados de Algoritmos I

Ano letivo 2020/2021

Relatório

4º Trabalho – Corrector Ortográfico

Docente:
Professora Lígia Ferreira

Trabalho realizado por:
Ludgero Teixeira nº41348

Évora, janeiro de 2021

Descrição de trabalho

Para a realização deste trabalho, foi pedido a implementação de um " **Corrector ortográfico"** usando as estruturas de dados **HashTables.**

Existe um **dicionário** previamente disponibilizado que permite verificar se as palavras existentes num ficheiro de texto estão correctas ou não
O objetivo deste trabalho é que o **corretor ortográfico** receba um ficheiro de texto como input e listar todas as palavras mal escritas no mesmo.

Nessa listagem deve-se figurar a linha do texto em que a palavra errada é encontrada, sendo que em cada palavra mal escrita o corrector deve sugerir correções.

Essas mesmas correções devem ser obtidas através das consultas ao **dicionário**, aplicando uma das seguintes regras:

- Adicionar uma letra
- Remover uma letra
- Trocar duas letras consecutivas

Classes a implementar

Funcionalidade	Interface	Implementação
Hash tables com endereçamento	_	Hashtable.java
Endereçamento linear	_	LinearHashtable.java
Endereçamento quadrático	_	QuadHashTable.java
Corrector Ortográfico	_	Corrector.java

Abordagem / Implementação

Neste trabalho forem implementadas as seguintes classes:

- HashTable.java
- LinearHashTable.java
- QuadHashTable.java
- Corrector.java

HashTable

Funções:

- **Insere**

  Nesta função é verificado se o **load factor** excede o valor 0.6 ,caso o mesmo se verifique, a função vai chamar o método **" rehash() "** seguidamente é feito uma inserção do novo elemento na hashtable. A variável **"usadas"** incrementa para que a quantidade de entradas preenchidas na **hashtable** esteja sempre atualizada a cada inserção.

  Complexidade temporal: O(n).

```
public void insere(AnyType x) {
    float carga = LoadFactor();
    if (carga > 0.6) {
       rehash();
    }
    int alocar = acesso(x);

    space[alocar] = new hashElement<>(x);
    usadas++;
 }
```

- **remove**

  Esta função procura se o elemento na **hashtable**, se houver uma posição na **hashtable** que contenha o elemento, o elemento contido nessa posição é removido.

  Complexidade temporal : O(n).

```
public void remove(AnyType x) {
    int pos = acesso(x);
    if (space[pos].element.equals(x)) {
       space[pos].remove();
    }
 }
```

- **rehash**

  Esta função cria uma nova **hashtable** como o dobro do tamanho da **hashtable** anterior, voltando a inserir os elementos que já se encontravam na tabela "antiga" na nova **hashtable**.

```
public void rehash() {
   int novoTamanho = novoTamanho(tamanho * 2);

   hashElement old[] = space;
   space = new hashElement[novoTamanho];
   tamanho = novoTamanho;
   usadas = 0;

   for (hashElement<AnyType> i : old) {
      if (i != null) {
         insere(i.element);
      }
   }
}
```

- **procurar**

  Esta função procurar, através de um tipo de acesso (linear ou quadrático) um elemento na **hashtable** se não houver nenhuma entrada na **hashtable** com esse elemento a função retorna **null** ,caso isso não se verifique retorna o elemento.

  Complexidade temporal : O(n).

```
public AnyType procurar(AnyType x) {
    int proc = acesso(x);
    if (space[proc] == null || !space[proc].existe) {
       return null;
    }
    return x;
 }
```

- **loadfactor**

  O load factor é uma medida do quão cheia esta uma **hashtable** até que seja preciso aumentar a sua capacidade.

  (o número de entradas preenchidas na tabela / o tamanho do array)

```
public float LoadFactor() {
   return (float) usadas / (float) tamanho;
}
```

LinearHashTable

Esta função **acesso(Anytype S)** é um tipo de inserção na **hashtable** (acesso linear).

O valor índex da entrada onde a chave vai ser inserida é calculado através da função hash : **s.hashCode()%tamanho**.

Caso haja uma colisão na inserção de uma chave, a entrada onde a colisão é criada é iterada linearmente (+1), até que seja encontrada uma entrada vazia.

Primeiramente a complexidade temporal irá ser O(1), mas após algumas inserções "blocos" de entradas preenchidas na **hashtable** iram começar a aparecer, neste caso a complexidade temporal irá ser O(n) onde n é o número de entradas existentes na **hashtable.**

```
    @Override
    protected int acesso(AnyType s) {
        int has = s.hashCode() % tamanho;

        if (has < 0) {
            has *= -1;
        }

        while (true) {
            if (space[has] == null) {
                return has;
            } else if (space[has].element.equals(s) && space[has].existe) {
                return has;
            }
            if (has + 1 == tamanho) {
                has = 0;
            } else {
                has++;
            }
        }

    }

}
```

QuadHashTable

Esta função **acesso(Anytype S)** é um tipo de inserção na **hashtable** (acesso quadrático).

O valor índex da entrada onde a chave vai ser inserida é calculado através da função hash : **s.hashCode()%tamanho**.

Caso haja uma colisão na inserção de uma chave, a entrada onde a colisão é criada é iterada pela seguinte forma (hash(s) + i^2), onde ” i “ é incrementado a cada tentativa de inserção após uma colisão.

Apesar de eliminar o “primary clustering” causado pelo LinearHashTable (acesso linear), este método de inserção gera outro tipo de problema, pois todos os elementos que **“hasham”** no mesmo sítio acessam as mesmas entradas.

A complexidade temporal é idêntica ao **“acesso linear”,** irá ser O(1), mas após algumas inserções a complexidade temporal irá ser O(n) onde n é o número de entradas existentes na **hashtable.**

```
@Override
 protected int acesso(AnyType s) {
    int has = s.hashCode() % tamanho;
    int inicialHas = has;

    if (has < 0) {
       inicialHas*=-1;
       has *= -1;
    }

    int i = 1;

    while (true) {
       if (space[has] == null) {
          return has;
       } else if (space[has].element.equals(s) && space[has].existe) {
          return has;
       }
       if (inicialHas + (i * i) >= tamanho) {
          has = inicialHas+(i*i);
          has=has%tamanho;

       } else {
          i++;
       }
    }


  }

}
```

Corrector

Na classe Corrector, é criada uma **HashTable** que vai conter todas as palavras contidas no ficheiro "**wordlist2020**" ou "**wordlist-big-20201212"**, que são "lidas" através de um **BufferReader**, sendo que de seguida, a inserção da mesma poderá ser feita através de acesso linear (**LinearHashTable**) ou acesso quadrático (**QuadHashTable**).

De seguida é lido o ficheiro input de texto "text.txt", onde sempre que uma linha desse ficheiro é lida, um contador que contem o número da linha que se esta a ler é incrementado. Um **StringBuffer** "opções" é criado que irá guardar temporariamente as possíveis alterações as palavras encontradas no ficheiro de input que não se encontram no dicionário. É também criado um array de strings "alfabeto" que vai conter todas as letras do alfabeto português (maiúsculas, minúsculas, com acentos).

Se uma palavra contida no ficheiro de input não estiver contida no "dicionário", poderão ser feitas "sugestões" de alteração a essa palavra de 3 maneiras...

Adicionar uma letra

```
if (dicionario.procurar(palavra) == null) {
            for (int i = 0; i < alfabeto.length; i++) {
               for (int x = 0; x <= palavra.length(); x++) {
                  opcoes = new StringBuffer(palavra);
                  opcoes.insert(x, alfabeto[i]);

                  if (dicionario.procurar(opcoes.toString()) != null) {
                     System.out.println(" |linha:" + NumeroDaLinha + " palavra: " + linha + "   sugestão --> " + opcoes);

                  }
               }
            }
```

Remover uma letra

```
for (int x = 0; x < palavra.length(); x++) {
               opcoes = new StringBuffer(palavra);
               opcoes.deleteCharAt(x);

               if (dicionario.procurar(opcoes.toString()) != null) {
                  System.out.println(" |linha:" + NumeroDaLinha +  " palavra: " + linha + "  sugestão --> " + opcoes);

               }
            }
```

Trocar duas letras consecutivamente

```
for (int x = 0; x < palavra.length(); x++) {
                    for (int y = 0; y < palavra.length(); y++) {
                       opcoes = new StringBuffer(trocar(palavra, x, y));

                       if (dicionario.procurar(opcoes.toString()) != null) {
                          System.out.println(" |linha:" + NumeroDaLinha +  " palavra: " + linha + "  sugestão --> " + opcoes);

                       }
                    }
                 }
```

O método " trocar" é utlizado na condição que oferece uma sugestão baseado na troca de duas letras consecutivas de uma palavra. Este método recebe como argumento uma string que irá ser a palavra vinda do texto.txt, e 2 variáveis que vão ser ter os seus valores " trocados", estas variáveis servem para percorrer a palavra, trocando pares de 2 caracteres ao longo que vai percorrendo a mesma.

Exemplo

Input contido no ficheiro de texto

```
relar
jugar
bolinhe
flor
violinou
Jouana
Almeirimy
Chamosca
Pertugel
jasmino
carle
toge
```

Output

```
-----------------------Corretor Ortográfico--------------------------
 |linha:1 palavra: relar   sugestão --> grelar
 |linha:1 palavra: relar   sugestão --> relvar
 |linha:1 palavra: relar  sugestão --> rela
 |linha:2 palavra: jugar   sugestão --> julgar
 |linha:3 palavra: bolinhe  sugestão --> boline
 |linha:5 palavra: violinou  sugestão --> violino
 |linha:6 palavra: Jouana  sugestão --> Joana
 |linha:7 palavra: Almeirimy  sugestão --> Almeirim
 |linha:11 palavra: carle  sugestão --> cale
 |linha:12 palavra: toge   sugestão --> togue
 |linha:12 palavra: toge  sugestão --> toe
```