
# Estrutura de Dados e Algoritmos I
# Relatório do Trabalho Prático nº2

Corretor Ortográfico

3ºsemestre 2016/2017

Docente: Prof. Lígia Ferreira

Alunos: João Dias nº35476

# Índice

# Introdução

No âmbito da unidade curricular de Estrutura de Dados e Algoritmos foi pedido que se implementasse um corretor ortográfico, utilizando tabelas de dispersão. Tabelas estas mais conhecidas por HashTables.

HashTables são um tipo de estrutura de dados que associa chaves de pesquisa a valores. Foi também pedido que implementássemos as respetivas tabelas: LinearHashTable, QuadraticHashTable e DoubleHashTable. Foram criadas estas três classes que derivarão da classe abstrata HashTable.

Com este trabalho também é pretendido que através de um ficheiro com um texto escrito se verifique que palavras estão escritas corretamente. Para verificar isto, é nos disponibilizado um ficheiro .txt com todas as palavras do dicionário. Por ultimo deve ser criado um ficheiro com o par erro, sugestão para o respetivo input.

# 1. Implementação Classe abstrata HashTable

A classe HashTable é uma classe abstrata pois os seus métodos vão ser herdados por outras classes. Esta classe também usa o Comparable para comparar elementos da tabela, quando se inserem ou procuram.

São usadas duas variáveis. A variável *elementos* que conta o numero de elementos presente na tabela e a variável tabela que é um array de AnyType.

Possui dois construtores. O primeiro não tem argumentos e cria o array tabela com um tamanho predefinido e o segundo cria o array com o tamanho recebido no argumento. Para alem disto estes construtores também inicializam o numero de elementos a zero.

Esta classe contem dois métodos abstratos (sem corpo) para procurar a posição correta de certo elemento na tabela e para procurar em que índice se encontra um certo elemento.

Também contem mais oito métodos para alem dos enunciados.

- O método *ocupados* retorna o numero de elementos presentes na tabela.
- O método *factorCarga* retorna um float que traduz a relação entre o numero de elementos na tabela e o espaço total.
- O método *makeEmpty* coloca todos os espaços da tabela com o valor null, apagando consequentemente a tabela.
- O método *remove* elimina um elemento recebido como argumento, fazendo uso de *procurar*.
- O método *insere* coloca numa posição correspondente um elemento, fazendo uso de *procPos*, e incrementa o numero de elementos.
- O método *rehash* com a ajuda de duas tabelas auxiliares vai aumentar em mais do dobro a tabela original. Faz isso duplicando o tamanho e incrementando uma unidade até obtermos um numero primo. Depois de aumentar o tamanho da tabela com a ajuda de um ciclo for e do método *insere* são colocados todos os elementos que estavam presentes na tabela antiga, mas em novas posiçoes.
- O método *isPrimo* como o nome indica vai verificar se o inteiro recebido como argumento é um numero primo, se for retorna true.
- Por ultimo o método *print* vai imprimir o conteúdo de todas as posições da tabela de forma ordenada.

# 2. Implementação Classe LinearHashTable

A classe LinearHashTable é uma das três classes que herdam os métodos da classe HashTable. Esta classe tem como objetivo a implementação de pesquisas e acessos lineares.

Em primeiro temos dois construtores iguais aos que encontramos na classe HashTable. Em seguida iremos ter dois métodos, um para retornar a posição onde colocar certo elemento e outro para procurar um elemento na tabela e retornar a sua posição.

- O método *procPos* é aquele que tem como objetivo verificar em que posição da tabela deve ser colocado o elemento segundo o respetivo HashCode. Como se trata de um acesso linear sempre que o HashCode originar um espaço onde já esta presente um elemento (colisão) irá ser incrementado o índice ate se encontrar um espaço vazio. Sempre que seja detetado através do método *factorCarga* que a tabela se encontra cheia, será feito um rehash e só depois será feito a nova procura de um índice.
- O método *procurar* como o nome indica irá localizar onde um respetivo elemento se encontra na tabela. Para isso iremos calcular o HashCode do elemento e através do método *compareTo* iremos verificar se o elemento na respetiva posição é o que procuramos. Caso não seja iremos procurar na posição a seguir. Caso percorramos a tabela toda e não encontremos o elemento chegaremos a conclusão que o elemento não se encontra na tabela.

# 3. Implementação Classe QuadraticHashTable

A classe QuadraticHashTable é em tudo semelhante à classe LinearHashTable, exceptuando o facto de as colisões serem tratadas de maneira diferente.

- No método *procPos* sempre que enfrentamos uma colisão, em vez de incrementarmos o índice, incrementamos o índice de forma quadrática. Ou seja, incrementamos o numero da tentativa ao quadrado (índice + i^2). Esta operação é realizada até encontramos um espaço null. O principal problema deste acesso é que todos os elementos que "hasham" no mesmo sitio, acedem às mesmas alternativas.
- No método *procurar* usa-se a mesma técnica que no método anterior e usa-se também como auxiliar o método *compareTo* para verificar quando dois elementos são iguais.

# 4. Implementação Classe DoubleHashTable

Por ultimo a classe DoubleHashTable é a ultima que herda os métodos da classe HashTable. É em tudo semelhante as duas classes tratadas anteriormente mas as colisões são tratadas através do uso de uma nova função hash.

- O método *hash2* faz a função que codifica pela segunda vez o elemento sempre que necessário. Este método recebe o elemento e verifica qual o numero primo logo a seguir ao tamanho da tabela. Esta função será *maxPrimo – elemento.hashCode mod maxPrimo.*
- O método *maxPrimo* simplesmente incrementa o inteiro recebido como argumento ate este numero ser primo e depois retorna-o.
- Nos dois métodos seguintes o tratamento de colisões consiste no aumento do índice consoante o resultado da nova função de hash.

# 5. Implementação Classe SpellChecker

A classe SpellChecker será como que o main do nosso programa e será aqui tratado os ficheiros correspondentes ao dicionário, ao input e aos erros-sugestões. Como variáveis desta classe teremos duas hashtables, uma para as palavras do dicionário e outra para as sugestões das palavras erradas. Foi escolhido tabela linear para o dicionário e quadrático para as sugestões.

## a. Classe SpellChecker

Nesta classe que serve como construtor é criado a hashtable linear onde serão guardadas todas as palavras lidas do ficheiro do dicionário, através de um bufferedReader. Tambem é criado a hashtable quadrática onde irão ser guardadas todas as sugestões para possíveis erros.

## b. Classe SpellCheck

O objetivo principal desta classe será ler o ficheiro input palavra por palavra e verificar se essa palavra esta contida na tabela do dicionário. Caso esteja é considerado que a palavra está bem escrita. Caso não esteja a palavra ira sofrer varias alterações (*adjacente*, *ad_char* e *rem_char*) e se algumas das alterações estiver contida na tabela do dicionário, é adicionada à tabela das sugestões.

## c. Classe geraFicheiro

Nesta classe pretende-se ler todas as palavras contidas na tabela das sugestões e escreve-las num ficheiro criado para guardar as sugestões dos erros.

# Conclusão

Com a realização deste trabalho posso concluir que adquiri e pus em pratico novos conhecimentos relacionados com a estrutura de dados HashTable.
Concluo também que nem todos os objetivos foram atingidos, pois ao verificar o ficheiro de inputs apenas crio as sugestões para os erros e não um ficheiro com o par erro → sugestão como era pedido no enunciado.
Tirando este aspecto o programa corre bem principalmente na criação das tabelas e no modo como lida com as colisões.