UNIVERSIDADE DE ÉVORA

4º Trabalho de Estruturas de Dados e Algoritmos

André Rato nº45517

25 de janeiro de 2021

Base de Dados: Professora Lígia Ferreira

Índice

Objetivos

Pretende-se o uso e implementação de tabelas de dispersão. Para isso, os alunos devem apresentar implementações de tabelas de dispersão com diferentes tipos de endereçamento. Essas implementações devem servir de suporte à elaboração de um corretor ortográfico. A utilização de classes do *package* "java.util" sem autorização do professor, com exceção de leitura de *input* e escrita de *output*, não é permitida.

Introdução

Uma tabela de dispersão, ou *hash table*, é uma estrutura de dados que associa chaves de pesquisa a valores. São tipicamente utilizadas para a indexação de grandes volumes de informação (como bases de dados). A sua implementação típica procura uma função de dispersão que seja de complexidade *O(1)*, não importando o número de dados na tabela (desconsiderando colisões). Em relação a outras estruturas de dados associativas, como um *array* simples, o ganho passa a ser maior conforme a quantidade de dados aumenta.

A função de dispersão, ou função *hash*, é a responsável por gerar um índice a partir de uma determinada chave. Caso a função seja mal escolhida, a tabela terá um mau desempenho. O ideal para a função de espelhamento é que sejam sempre fornecidos índices únicos para as chaves de entrada. Quando isto não acontece, ou seja, quando dois valores geram o mesmo índice, ocorre uma colisão.

Existem vários mecanismos criados para evitar ou resolver problemas relacionados a colisões, como o endereçamento aberto e o encadeamento.

No método de endereçamento aberto, os registos em conflito são armazenados dentro da própria tabela de dispersão. A resolução das colisões é realizada através de buscas padronizadas dentro da própria tabela. As tabelas que utilizam este método podem necessitar de redimensionamento. O endereçamento aberto possui três estratégias: linear (incremental), quadrática e dispersão dupla.

Para a estratégia linear, é utilizada uma segunda função matemática para calcular a posição em que deve ser feito o próximo teste, a função de "redispersão".

$$rh(pos, tentativas) = (pos + tentativas)\ mod Ttabela$$

Desta forma, eliminamos o agrupamento primário, porém geramos outro fenómeno conhecido como agrupamento secundário. Ocorre que as chaves de valores diferentes e mesmo valor de dispersão possuem a mesma sequência de testes.

Para a estratégia quadrática, é utilizada a seguinte função:

$$rh(pos, tentativas) = (pos+tentativas^2)\ mod Ttabela$$

Para reduzir o agrupamento primário, procura-se por um lugar livre através da fórmula: $h + n2$, em que h representa o índice de colisão e n o sequencial de busca pela nova posição.

Para a estratégia de dispersão dupla:

$$rh(ch, tentativas) = (h2(ch) + tentativas)\ mod Ttabela$$

Utilizamos então a função *h2(ch)*:

$$h2(ch) = 1 + ch\ mod(Ttabela - 1)$$

No método de encadeamento, a informação é armazenada em estruturas encadeadas fora da tabela de dispersão. Encontra-se uma posição disponível na tabela e indicamos que esta posição é a próxima a ser procurada.

A tabela de dispersão é uma estrutura de dados do tipo dicionário, que não permite armazenar elementos repetidos, recuperar elementos sequencialmente (ordenação), nem recuperar o elemento antecessor e sucessor. É necessário conhecer a natureza da chave a ser utilizada para que a otimização da função de dispersão seja muito eficiente. No pior dos casos, a ordem das operações pode ser *O(N)*, caso em que todos os elementos inseridos colidem.

Implementação das tabelas de dispersão

Para a implementação das tabelas de dispersão foram utilizadas quatro classes em Java:

- HashTableElement<AnyType>:
  - variáveis de classe:
    - *AnyType element*;
    - *boolean exists*: valor de true caso o elemento esteja presente na tabela.
  - métodos da classe:
    - *HashTableElement(AnyType element)*;
    - *void remove()*: remove o elemento da tabela colocando o valor de *exists* como falso (*lazy delection*);
    - *String toString()*.
- HashTable<AnyType>:
  - variáveis de classe:
    - *HashTableElement<AnyType>[] elements*: array responsável por guardar os valores da tabela;
    - *int size*: tamanho da tabela;
    - *int used*: número de lugares da tabela que estão ocupados.
  - métodos da classe:
    - *HashTable()*: cria uma tabela vazia de tamanho 11;
    - *HashTable(int size)*: cria uma tabela vazia de tamanho *size*;
    - *int used()*: retorna o valor presente na variável *used*;
    - *double loadFactor()*: calcula e retorna o fator de carga da tabela;
    - *abstract int searchPosition(AnyType element)*;
    - *void allocateTable(int size)*: cria uma nova tabela vazia de tamanho *size*;
    - *void makeEmpty()*:limpa a tabela de dispersão, tornando-a vazia;
    - *AnyType search(AnyType element)*: procura o elemento desejado na tabela e, caso não o encontre, retorna null;
    - *void remove(AnyType element)*: procura o elemento desejado na tabela e, caso o encontre, remove-o;
    - *void insert(AnyType element)*: calcula o fator de carga da tabela, "redispersando" a tabela se o resultado for superior a 0.5 e, após isso, coloca o *element* na tabela, incrementando a variável *used*;
    - *void reHash()*: cria uma nova com um tamanho superior ao inicial, inserindo os elementos da tabela inicial na nova tabela;
    - *int newSize(int size)*: calcula o novo tamanho da tabela duplicando o valor inicial e tentando sempre manter o tamanho da tabela primo;
    - *void print()*: imprime a tabela de dispersão;
    - *int unityLength(int number)*: calcula o número de algarismos de um número;
    - *String getSpaces(int number)*: retorna uma string com o número de espaços correspondente à diferença entre o número de algarismos do *size* da tabela e o número de algarismos do argumento (*number*).
- LinearHashTable<AnyType> extends HashTable<AnyType>:
  - métodos da classe:
    - *int searchPosition(AnyType element)*: calcula qual a posição correspondente ao *element* (no caso da estratégia linear).
- QuadracticHashTable<AnyType> extends HashTable<AnyType>:
  - métodos da classe:
    - *int searchPosition(AnyType element)*: calcula qual a posição correspondente ao *element* (no caso da estratégia quadrática).

Implementação do corretor ortográfico

Para a implementação do corretor ortográfico, foram utilizados dois ficheiros:

- wordlist-2020.txt: ficheiro que contém 998649 palavras portuguesas;
- texto.txt: ficheiro que contém o texto que será analisado e ao qual serão feitas sugestões de correção às palavras identificadas como erro.

Foi utilizada uma tabela de dispersão linear (*dictionary*), pois ao usar uma estratégia quadrática, não há garantia de encontrar uma posição vazia, uma vez que mais da metade da tabela se encontra cheia, já que as colisões são tratadas usando apenas metade da tabela.

Relativamente à leitura do primeiro ficheiro, foram utilizadas classes e funcionalidades do package *java.io*, tais como as classes *File*, *FileReader* e *BufferedReader* e os métodos *readLine()* e *close()*. Nessa leitura foi utilizada a técnica LBL (*line by line*), que consiste na leitura e análise do ficheiro, linha a linha.

Para a leitura do segundo ficheiro, o package *java.io* foi utilizado, juntamente com o package *java.util*, para que o utilizador consiga definir o caminho/nome do ficheiro de texto. É utilizada também uma variável de controlo (*boolean control*) para que o programa saiba quando o caminho/nome do ficheiro de texto é válido. A técnica LBL é aplicada também para a leitura e análise do segundo ficheiro.

Cada linha do ficheiro de texto passa por um conjunto de processos até que seja gerado o output da mesma:

1. as várias palavras da linha são separadas e guardadas num array;
2. os sinais de pontuação são removidos da palavra que está a ser analisada;
3. a palavra é pesquisada na tabela de dispersão e, caso a pesquisa falhe (seja retornado *null*), é iniciado o processo de apresentação de sugestões;
4. utilizando todas as letras do alfabeto português (considerando o hífen e os caracteres acentuados), estas são adicionadas, uma a uma, a palavra que está a ser analisada, nas várias posições;
5. as novas palavras são procuradas na tabela hash e, caso alguma seja encontrada, é considerada como sugestão;
6. são removidas todas as letras, individualmente, da palavra inicial;
7. as novas palavras são procuradas na tabela hash e, caso alguma seja encontrada, é considerada como sugestão;
8. são trocadas duas letras consecutivas da palavra inicial, sucessivamente;
9. as novas palavras são procuradas na tabela hash e, caso alguma seja encontrada, é considerada como sugestão;
10. as várias letras da palavra são trocadas, individualmente, por outras letras;
11. as novas palavras são procuradas na tabela hash e, caso alguma seja encontrada, é considerada como sugestão;
12. se não tiver sido encontrada nenhuma solução, o corretor indica o erro, apresentando a mensagem "*Can´t give suggestion*".

Durante o processamento das palavras, são utilizados quatros métodos estáticos da classe *Corretor*:

- *String addLetter(String word, char c, int index)*: método utilizado para adicionar uma letra (*c*) à palavra (*word*) numa certa posição (*index*);
- *String removeLetter(String word, index)*: método utilizado para remover a letra de uma certa posição (*index*) da palavra (*word*);
- *String switchNextLetter(String word, int index)*: método utilizado para trocar a letra de uma certa posição (*index*) com a seguinte de uma palavra (*word*);
- *String changeLetter(String word, char c, int index)*: método utilizado para trocar a letra de uma certa posição (*index*) com outra letra (*c*).

Exemplo de output

- Input:

```
mensagem de test.
vamos verificar se há algmu erro.
se houver basta só identificari!
certu?
aeiouoiea!
```

- Output:

```
Dictionary file opened successfully!
Closing dictionary file!
File path of the text file: src-textfiles/text.txt
Text file opened successfully!

Line 1: test » testa (add a letter)
Line 1: test » teste (add a letter)
Line 1: test » testo (add a letter)
Line 1: test » tesa (change letters)
Line 1: test » tese (change letters)
Line 1: test » teso (change letters)
Line 2: algmu » algum (switch letters)
Line 3: identificari » identificaria (add a letter)
Line 3: identificari » identificarei (add a letter)
Line 3: identificari » identificai (remove a letter)
Line 3: identificari » identificar (remove a letter)
Line 3: identificari » identificara (change letters)
Line 3: identificari » identificará (change letters)
Line 4: certu » certa (change letters)
Line 4: certu » certo (change letters)
Line 5: aeiouoiea » Can't give suggestion

Closing text file!
```

Referências bibliográficas

Ficha 9 de EDA1 - Moodle;
Enunciado do trabalho - Moodle;
Tabela de dispersão - Wikipédia;
Different ways of reading a text file in Java – GeeksforGeeks.