Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea 5: Corrector Ortográfico

Fecha: 28 de junio de 2017
Autor: Daniel Araya Poblete
Email: Daniel.arayap1@gmail.com
Curso: CC3001-2
Profesor: Nelson Baloian

Introducción

El problema a resolver en esta tarea es la implementación de un corrector ortográfico que identifique las palabras escritas de forma incorrecta y se las muestre al usuario. Una forma de realizar esto es mediante el uso del TDA Diccionario simplificado, que cuente sólo con las operaciones de inserción y búsqueda.

Dentro de las muchas posibles opciones para implementar este TDA Diccionario, en esta tarea se utilizarán dos implementaciones: usando un árbol AVL y usando un arbol digital de prefijos Trie.

Una vez que se tienen estas dos estructuras implementadas, se debe leer para cada implementación las palabras contenidas en un archivo de texto e insertarlas para armar la correspondiente estructura de datos.

Finalmente se crea un programa principal para cada implementación tal que al ingresar una frase lea cada palabra de esa frase, verifique si la estructura contiene la palabra y retorne las palabras que están escritas incorrectamente (es decir, que no estén contenidas en la estructura).

Análisis del problema

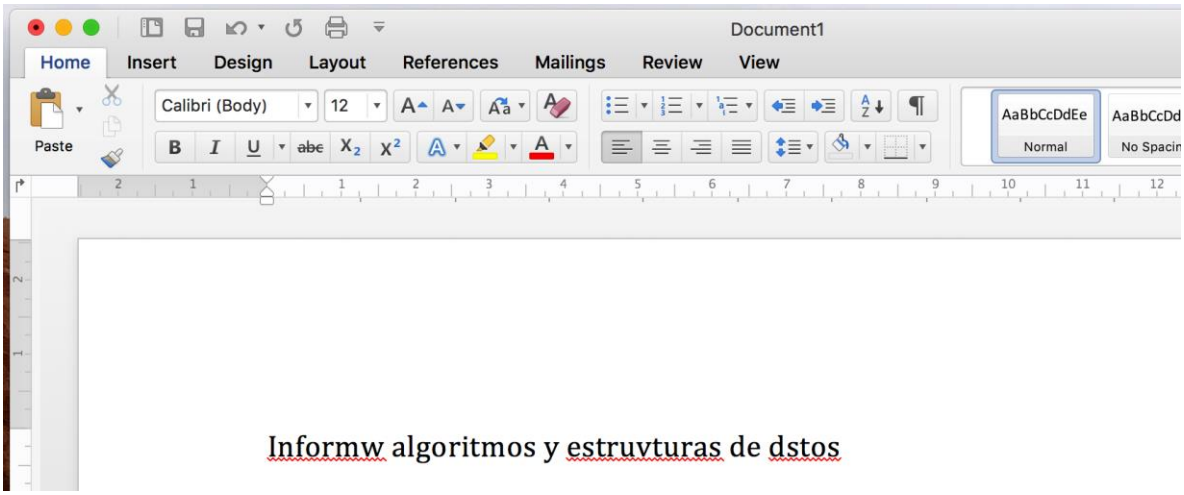

Figura 1. Ejemplo del corrector ortográfico de Word.

El TDA Diccionario que se desea implementar en esta tarea, debe contener las operaciones de inserción y búsqueda. Como se mencionó en la introducción, en esta tarea se implementaron dos versiones del TDA Diccionario:

- Utilizando un AVL arbol balanceado de busqueda binaria. Esto es, para cada nodo se cumple que:

$$|altura\ del\ subárbol\ izquierdo - altura\ de\ subarbol\ derecho| \leq 1$$

  Ademas de esto, se pide que las operaciones de *insertar* y *contiene* se realicn de forma similar al caso de un ABB (cuidando que no se rompa la propiedad de arbol balanceado), comparando las palabras según el orden lexicográfico (ej. Dedo>dado, pato<perro, etc.).

- Utilizando un Trie, tal que cada nodo del arbol contiene un arreglo de nodos hijos donde cada elemento corresponde a un nodo que contiene un carácter. Para esto se sugiere utilizar arreglos de tamaño 33, donde se incluyen los 26 caracteres a-z, los 5 caracteres á-ú, además de la ñ y un carácter especial que marque el fin de una palabra en el Trie (‘$’).

Como requerimiento fnal de esta tarea, se debió realizar una limpieza para cada palabra antes de insertarla en la estructura de datos y también para cada palabra contenida en la frase que ingrese el usuario del programa. La limpieza consistió en pasar cada letra de la palabra a minúscula, y eliminar cualquier dígito, signo de puntuación o carácter especial contenido en la palabra.

Solución del problema

Para esta tarea se crearon las clases AVL y Trie.

Para la clase AVL se trabaja con un NodoAVL con los atributos palabra(String), altura(int), y los hijos izq y der (NodoAVL). Se debían implementar los métodos insertar y contiene, pero además al ser un arbol balanceado, se debieron crear los métodos de rotaciones simples y dobles tanto a la izquierda como a la derecha (ver archivo AVL.java)

Se crearon dos métodos *insertar,* uno que recibe la palabra a insertar en el árbol y el NodoAVL del árbol en el que se quiere insertar la palabra, y otro método homónimo que recibe unicamente la palabra y la inserta en la raíz de árbol AVL. El método de inserción se realiza tal como se presenta en el enunciado de la tarea, es decir, se inserta como si fuera un ABB (respetando el orden lexicográfico) y se realizan rotaciones si se rompe la condición de equilibro en el arbol (i.e si la diferencia de alturas en los nodos es mayor a 1).

De la misma manera se crearon dos métodos *contiene,* uno que recibe la palabra que se quiere verificar y el NodoAVL del árbol en que se iniciará la búsqueda de la palabra; y el segundo *contiene* recibe únicamente la palabra y la busca en el NodoAVL raíz. El método de búsqueda es idéntico al de un ABB, es decir, si la palabra es menor lexicográficamente se busca en el hijo izquierdo y si es mayor, la búsqueda se realiza en el hijo derecho.

Para la clase Trie se trabaja con un NodoTrie que cuenta con un arreglo *hijos* cuyas 32 componentes son Nodotrie y una variable *fin* (booleana), ya que finalmente se escogió utilizar una variable de tipo *boolean* que determine si el carácter contenido en el NodoTrie corresponde al final de una palabra.

En el método *insertar* cada cáracter en la palabra se evalúa si el nodo con ese carácter existe en el arreglo de hijos del NodoTrie. Si no existe, se inserta un nuevo nodo con el carácter en la ubicación que le corresponda en el arreglo (ver Figura 2). Si el nodo existe, el puntero ahora apunta al nodo hijo en el índice que corresponde a ese carácter.

a	B	c	d	e	f	g	h	i	j	k	l	m	n	o	p	Q	r	s	t	u
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
v	W	x	y	z	á	é	í	ó	ú	ñ										
21	22	23	24	25	26	27	28	29	30	31										

Figura 2. Ubicación de los caracteres en el arreglo *hijos*.

El método *contieneNodo* entrega el NodoTrie que contiene la letra final de la palabra a buscar, y en caso de no encontrarla retorna *null.* El método *contiene,* por lo tanto, sólo aplica el método *contieneNodo* a la palabra a buscar y verifica que el resultado no sea nulo y, de ser así, que el nodo contenga efectivamente la letra final de la palabra.

Modo de uso

En la entrega de esta tarea encontrará, aparte del presente informe, los archivos Trie.java y AVL.java. Ambos tienen un método *main()* que al ser ejecutado, solicitará al usuario una frase para evaluarla y, tal como se muestra en el ejemplo del enunciado, imprimirá en pantalla las palabras que no están escritas correctamente (que no aparecen en la estructura de datos.

El archivo espanol2.txt debe estar en la misma carpeta *src* que los archivos Trie.java y AVL.java para que estos puedan leerlo.

Resultados

En el AVL la inserción es de O(log(N)), sin correr el riesgo de que tienda a O(N) porque al ser balanceado, no se generan desequilibrios que conviertan al árbol en una lista enlazada. El método *contiene* es una búsqueda en el árbol AVL, por lo que éste también es de O(log(N)). El orden de inserción y búsqueda en el AVL no depende ni de P(largo de la palabra) ni del tamaño del alfabeto σ.

Por otro lado, en el árbol Trie, estos métodos operan de distinta manera y por lo tanto sus órdenes de tiempo son distintos a los del AVL. En el Trie tanto *insertar* como *contiene* son métodos de orden O(|P|) donde |P| es la cantidad de caracteres que tenga la palabra.

El tamaño en memoria del AVL es de O(N), la cantidad de palabras en el archivo, y en el Trie esto es O(N*|P|).

.