Práctica 8: Algoritmos sobre palabras

Aclaración: se usan como sinónimos los términos "*string*", "palabra", "cadena", "texto" y "cadena de texto". Una palabra no es otra cosa que una lista símbolos de algún alfabeto. Los símbolos a veces también se llaman "letras" o "caracteres".

Ejercitación básica

Ejercicio 1. (Árboles digitales). Consideremos el alfabeto $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ de los dígitos decimales. Dado un número entero $n \geq 0$, podemos pensarlo como una palabra sobre el alfabeto Σ si lo escribimos en su representación decimal, sin 0s a la izquierda.

Diseñar e implementar en Python un diccionario cuyas claves sean números enteros no negativos, con las siguientes operaciones:

a) Insertar una clave numérica, asociándola a un valor.

b) Buscar una clave numérica, determinando si está presente y encontrando el valor que tiene asociado, si lo hay.

El diccionario debe estar implementado sobre un **trie** sobre el alfabeto Σ. ¿Cuáles son las complejidades de las operaciones? ¿Cómo se relacionan con los costos de insertar y buscar en un AVL?

Ejercicio 2. Se tiene un texto fijo de longitud n sobre el que se quieren hacer muchas consultas. Cada consulta consiste en buscar una palabra de longitud entre 1 y 10 para determinar si aparece o no en el texto. Proponer una estructura de datos que permita:

a) Preprocesar el texto, construyendo una estructura de datos auxiliar en tiempo $O(n)$ en peor caso.

b) Consultar si una palabra aparece en el texto, aprovechando la estructura de datos auxiliar, en tiempo $O(1)$ en peor caso.

Ejercicio 3. Diseñar e implementar en Python un algoritmo que reciba como entrada un diccionario representado sobre un trie, y devuelva como salida la lista de todas las claves definidas en el diccionario.

Ejercicio 4. Sean Σ^* un alfabeto y $w_1, \ldots, w_n \in \Sigma^*$ palabras sobre ese alfabeto. Escribimos $|w|$ para denotar la longitud de una palabra w. Llamamos L a la longitud total del texto, es decir $L = \sum_{i=1}^{n} |w_i|$. Diseñar un algoritmo para determinar si hay palabras repetidas en la lista con complejidad temporal $O(L)$ en peor caso.

Ejercitación adicional

Ejercicio 5. Se quiere diseñar una estructura de datos que represente un diccionario cuyas claves son *strings*. Las operaciones de inserción, búsqueda y eliminación son las usuales. Se desea agregar una operación `#mayores`(d, w) que, dado un diccionario d y una palabra w, determine la cantidad de claves del diccionario d que son estrictamente más grandes que x. Las palabras se comparan de acuerdo con el orden lexicográfico, es decir, el orden usual del diccionario. Por ejemplo:

$$\texttt{abejorro} < \texttt{vaca} < \texttt{vacaciones} < \texttt{viento} < \texttt{y}$$

Si el diccionario d contiene como claves a las 5 palabras de arriba, entonces `#mayores`(d, "`verbo`") $= 2$. Proponer una modificación en la estructura del trie y en los algoritmos que permitan implementar la operación `#mayores`(d, w) en tiempo $O(m)$ en peor caso, donde $m = |w|$ es la longitud de la palabra en cuestión.

Ejercicio 6. Se tiene una lista de palabras fija $\{w_1, \ldots, w_k\}$ sobre la que se quieren hacer muchas consultas. Sabemos que cada palabra es de longitud entre 1 y 10. Cada consulta consiste en recibir un texto de longitud n y determinar si **alguna** de las palabras de la lista $w_1, \ldots, w_k$ aparece en el texto al menos una vez. Proponer una estructura de datos que permita:

a) Preprocesar la lista de palabras, construyendo una estructura de datos auxiliar en tiempo $O(k)$.

b) Recibir un texto de longitud n y determinar si alguna de las palabras aparece en el texto, usando la estructura de datos auxiliar, en tiempo $O(n)$ en peor caso.

Ejercicio 7. Considerar la siguiente variante del ejercicio anterior: dado un texto de longitud n, se quiere determinar ahora si **todas** las palabras de la lista $w_1, \ldots, w_k$ aparecen en el texto al menos una vez. Mostrar que es posible hacerlo con las mismas complejidades del ejercicio anterior.