ESTRUCTURA DE DATOS Y ALGORITMOS 2
SEGUNDA ENTREGA DE EJERCICIOS

Para la realización de este trabajo la cátedra provee un proyecto de Visual Studio con las funciones a implementar, algunos datos de prueba y el framework visto en clase.

IMPORTANTE:

1. La definición de cada función incluye siempre indicar sus **pre** y **postcondiciones**.
2. Las pruebas provistas por la cátedra constituyen meros ejemplos y no son para nada exhaustivas. Se espera que los alumnos desarrollen pruebas propias para su código.
3. Debe entregarse solamente una carpeta comprimida conteniendo el proyecto “Obligatorio”, sin el Framework.

?1. Hashing

En esta tarea vamos a implementar un programa que imprima todos los anagramas de una cadena. Dos cadenas son anagramas si al reordenar las letras de una se puede formar la otra. Por ejemplo, “nacionalista” y “altisonancia” son anagramas. Para los propósitos de esta entrega estamos interesados sólo en los anagramas de una cadena que aparecen en el diccionario. El diccionario que se debería usar está disponible aquí [1].

Algoritmo e implementación. Ya que vamos a estar realizando múltiples consultas sobre anagramas, lo primero que debemos hacer es cargar todas las (80,383) palabras del diccionario en una estructura de datos adecuada. Una forma que utilizaremos para hacer esto es primero ordenar las letras de cada palabras que queramos insertar en la estructura, de forma de producir una clave para cada palabra. Por ejemplo, la clave para la palabra “estudio” sería “deiostu” [2]. Aquí notamos que todas las palabras que son anagramas tienen la misma clave, creando una asociación *clave* ↔ *[anagramas]*. Una forma prolija de representar esta situación es utilizando el TAD Tabla.

clave	**[anagramas]**
aaaciilnnost	altisonancia, nacionalista
amor	roma, ramo, maro, amor, mora
...	...

[1] https://raw.githubusercontent.com/javierarce/palabras/master/listado-general.txt
[2] Esto también se conoce como el alfagrama de una palabra.

Un requisito importante es que el usuario debe ser capaz de buscar anagramas de una palabra de manera eficiente. Utilizaremos *hashing* para implementar la Tabla y luego cargarla con todas las palabras del diccionario. Una vez que la tabla se encuentre cargada, buscar los anagramas de una palabra consiste en obtener la lista dada la clave.

Detalles. Puede crear su tabla de hash del tamaño que lo desee pero siguiendo las recomendaciones de los factores de carga para hash abierto y cerrado. Para debuggear su código recomendamos que trabaje con un diccionario de práctica más pequeño, por ejemplo con 10 palabras, y una tabla también más pequeña. Recuerde que está bien trabajar con el factor de carga y eventualmente sacrificar espacio por performance, de eso es de lo que se trata el hashing.

Debe realizar dos implementaciones del TAD Tabla usando ambas técnicas de resolución de colisiones: una implementación de hash abierto y otra cerrado, e implementar las funciones:

```
Puntero<Tabla<C, V>> CrearTablaHashAbierto(nat cantidadElementos,
                                           Puntero<FuncionHash<C>> fHash,
                                           const Comparador<C>& comp)
```

y

```
Puntero<Tabla<C, V>> CrearTablaHashCerrado(nat cantidadElementos,
                                           Puntero<FuncionHash<C>> fHash,
                                           const Comparador<C>& comp)
```

El atributo cantidadElementos hace referencia a N en la ecuación del factor de carga, donde B es la cantidad de cubetas:

$$\lambda = \frac{N}{B}$$

Puede ignorar cualquier palabra del diccionario que contenga caracteres de puntuación. Adicionalmente, puede convertir cualquier caracter en mayúscula a minúscula. Es decir, que sólo representamos las palabras que contienen caracteres en minúscula.

Debe implementar la función `Array<Cadena> Anagramas(const Cadena& c)` que dada una cadena retorna todos los anagramas de dicha cadena que se encuentran en el diccionario. Su función deberá lograr esto en O(1) caso promedio, por lo que se utiliza la implementación con hashing. Documente cualquier decisión de diseño del algoritmo y comente por qué su algoritmo cumple con el orden especificado.

?2. AVL.

Recordemos que un AVL es un árbol binario que se encuentra balanceado, es decir que para cada nodo la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha. Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se ha de realizar de una forma especial. Si al realizar una operación de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de rotaciones de los nodos. Estas rotaciones pueden ser *simples* o *dobles*.

Se pide implementar la siguientes función `bool EsAVL(Puntero<NodoArbol<T>> raiz, const Comparador<T>& comp)` que dado un árbol binario (no necesariamente ordenado) y un comparador determina si el árbol es un AVL.