ESTRUCTURA DE DATOS Y ALGORITMOS 2
SEGUNDA ENTREGA DE EJERCICIOS

Se aceptan entregas por Aulas hasta el 25/09/2016 a las 23:55

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. Un requisito importante es que el usuario debe ser capaz de buscar anagramas de una palabra de manera eficiente. 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]. Utilizaremos una tabla de hash para guardar pares de cadenas, donde el par consiste de la palabra original y su clave. Para esto quizás pueda resultar útil usar la clase Tupla del Framework o simplemente crearse una pequeña clase para representarlo.

Al realizar una inserción en la tabla de hash, computaremos el hash de la clave de la palabra para calcular la cubeta correcta (i.e. el lugar de la tabla de hash en el que insertaremos). Esta implementación garantiza que todas las palabras que son anagramas de una determinada palabra, se guardan en la misma cubeta de la tabla de hash. De manera

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

similar, al buscar un anagrama, primero computamos la clave de la palabra que estamos buscando y luego el hash de la clave; finalmente, sólo debemos buscar los anagramas en la cubeta correspondiente. Siéntase libre de usar los métodos descritos en la bibliografía y en clase para escribir una función de hash adecuada para una cadena (pero por favor cite cualquier fuente que use). Asimismo, de ser posible asegúrese de que su función de hash sea eficiente y no relacione dos conjuntos de anagramas completamente distintos en la misma cubeta. En caso que su función de hash tenga este comportamiento, deberá resolver las colisiones de alguna manera que usted encuentre conveniente.

Detalles. Puede crear su tabla del tamaño que lo desee pero es recomendable que tenga al menos 80,383 cubetas. 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 el tamaño de la tabla debería ser un número primo, de forma de reducir las colisiones. También recuerde que está bien sacrificar espacio por performance, de eso es de lo que se trata el hashing. Sin embargo, su tabla no debería superar las 700,000 cubetas.

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 realizar dos implementaciones de la tabla hash 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 cubetas,
                                           const FuncionHash<C>& fHash,
                                           const Comparador<C>& comp)
```

y

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

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. 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.