Proyecto 2: Corrector ortográfico

Estructuras de Datos, Primer Semestre 2016

Prof. Diego Seco

Ayudantes: Diego Gatica y Paulo Olivares

1. Descripción del Problema

En este proyecto se pide implementar un corrector ortográfico simple para el español o inglés. El corrector ortográfico incorporará una base de datos de palabras, y será construido a partir de una lista de palabras, organizando la información de tal forma que se pueda hacer una búsqueda eficiente.

Las palabras a ser corregidas se leen, una por una, de un archivo de entrada (de texto) y son entregadas al corrector ortográfico para su procesamiento. Si el corrector ortográfico encuentra una coincidencia exacta, éste simplemente indica que la palabra está escrita correctamente. Si no es así, el corrector ortográfico retornará una lista (posiblemente vacía) de sugerencias ortográficas. Para seleccionar las sugerencias, el corrector ortográfico aplica reglas para decidir si una palabra es candidata a ser la forma correcta de escribir la palabra. Supongamos que M es la palabra cuya ortografía se está comprobando, y que una palabra C es candidata a la lista de palabras sugeridas. El corrector ortográfico considerará que C es una sugerencia para M aplicando las siguientes reglas (en el orden dado):

- Si las longitudes de M y C difieren en más de dos, entonces C no se sugiere.
- Si M es un prefijo de C, o si C es un prefijo de M, entonces se sugiere C.
- Comparar, uno a uno desde el comienzo, los caracteres de M y C hasta terminar uno o ambos strings (lo que ocurra primero). Si hay a lo más dos diferencias, y ambos M y C son de largo mayor a tres, entonces sugiera C.

Estas reglas son un tanto arbitrarias y demasiado simples para un corrector ortográfico real. Sin embargo son aceptables para esta tarea donde el objetivo principal es el diseño de la estructura de datos y no el algoritmo de comparación. Reglas más sofisticadas podrán ser bonificadas con crédito extra.

2. Implementación

El programa debe ser llamado por línea de comandos recibiendo como parámetros: el archivo que contiene las palabras del diccionario, el archivo con el texto a revisar y un log con las correcciones. Una llamada al programa debe ser de la forma:

```
corrector <archivo_diccionario> <archivo_a_revisar> <log_revision>
```

El programa debe verificar la existencia de los dos primeros archivos de entrada e imprimir un error si alguno no existe. El contenido y formato de los archivo de entrada `archivo_diccionario` `archivo_a_revisar` y salida `log_revision` se encuentra detallado más abajo.

Existen diversas formas de implementar un corrector ortográfico. En este proyecto la implementación debe ajustarse a las siguientes restricciones:

- El diccionario de palabras debe almacenarse en un árbol especial denominado *trie* (ver Anexo A). Esta sencilla pero potente estructura de datos no se suele ver en el curso básico de Estructuras de Datos pero, dada su simplicidad y semejanza con cualquier otra implementación de árboles, la descripción incluida en el anexo debería ser suficiente para su implementación.
- El corrector ortográfico debe ser implementado como una clase. Contendrá un trie, pero no será simplemente un trie.
- Debe existir una clase "controladora"que se preocupe de procesar los archivos de entrada y de crear el archivo de salida. Esta clase contendrá un corrector ortográfico (y probablemente otras variables más).
- En la implementación se deben respetar los principios de buen diseño, abstracción y encapsulamiento. Por ejemplo, debe definir una interfaz para el corrector ortográfico, el árbol no debe asumir responsabilidades que no le pertenezcan como la lectura de palabras a revisar o la construcción de la lista de sugerencias de palabras alternativas, la representación debe ser privada y las modificaciones y consultas se deben hacer sólo a través de métodos.

3. Archivos

Los archivos de entrada y salida del programa son los siguientes:

(1) `archivo_diccionario`**:** archivo de entrada que contiene una palabra por línea. Puede contener comentarios partiendo con el símbolo %. Pueden encontrar ejemplos de diccionarios aquí: `http://www.winedt.org/dict.html`. Asuma que todas las palabras están con minúsculas.

(2) `archivo_a_revisar`**:** archivo que contiene el texto a revisar. El archivo puede contener puntuación y combinaciones de mayúsculas y minúsculas. Para poder hacer comparaciones válidas será necesario procesar el texto de tal forma de extraer cada palabra (delimitada por un espacio al principio y otro al final), luego reemplazar todas las mayúsculas por minúsculas y eliminar cualquier puntuación que haya al principio o al final. Por ejemplo si encuentra la palabra "Hola" revisará la palabra "hola". Si encuentra el texto "Claro!!!" revisará "claro".

(3) `log_revision`**:** este archivo de salida debe contener el log de la revisión ortográfica del `archivo_a_revisar`. Debe contener:

- Nombre del alumno, nombre del archivo que se revisó y diccionario utilizado para la revisión (URL en la cuál está disponible para descarga). Cada una de estas cosas debe estar impresa en una línea.
- Un listado donde cada fila incluya primero la palabra mal escrita y luego la lista de correcciones sugeridas para ella (en orden lexicográfico) desplegadas de la siguiente forma:

  `<palabra_mal_escrita>: <lista_de_sugerencias>`

  No se deben desplegar las palabras bien escritas. Es importante mantener este formato para poder automatizar parcialmente la corrección de la tarea.
- Un resumen de estadísticas de la revisión que incluya: número total de palabras que contiene el diccionario, número de palabras escritas correctamente y número de palabras con error. Cada una en una línea.

4. Evaluación

El proyecto se realizará en parejas. Enviar en piazza (mensaje privado a *Instructors*) lo siguiente:

1. Un informe que:
   a) Incluya portada, descripción de la tarea, descripción de la solución propuesta (puede emplear diagramas de clase para explicar la solución) y detalles de implementación.
   b) Sea claro y esté bien escrito. Un informe difícil de entender es un informe que será mal evaluado aunque todo esté bien implementado. La persona que revise el documento debe poder entender su solución sólo mirando el informe.
   c) Esté en formato pdf.
2. Un archivo comprimido con todos los ficheros fuente implementados para solucionar la tarea. El informe debe hacer referencia a ellos y explicar en qué consiste cada uno.

Fecha de Entrega: viernes 13 de Mayo 11:59PM

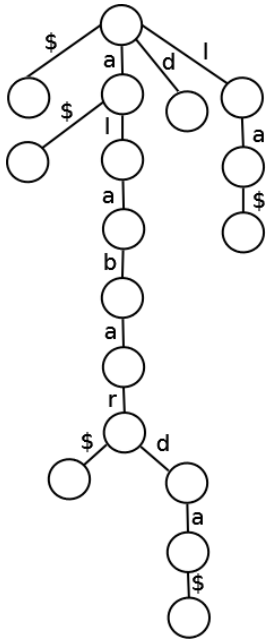
A. Trie

Un trie[1] permite almacenar un conjunto de strings $T_1, ..., T_k$ soportando búsquedas (predecesor/sucesor) en tiempo proporcional al largo del patrón buscado: $O(m)$.

¿Por qué no usar un BST más estándar?
El informe debe responder a esta pregunta, comparando ambas estructuras (a nivel teórico, no es necesario implementar/utilizar un BST).

Estructura: Árbol donde cada rama hija está etiquetada con un símbolo del alfabeto $\sum$.
Todos los strings deben estar terminados por un símbolo especial cuyo valor de codificación es menor a todos los otros símbolos de $\sum$. En este documento se usará el caracter $ para ejemplificar.
Todos los recorridos desde el nodo raíz hasta los nodos hojas corresponden a un string almacenado.

Ejemplo: {'alabar$', 'a$', 'la$', 'alabarda$' }

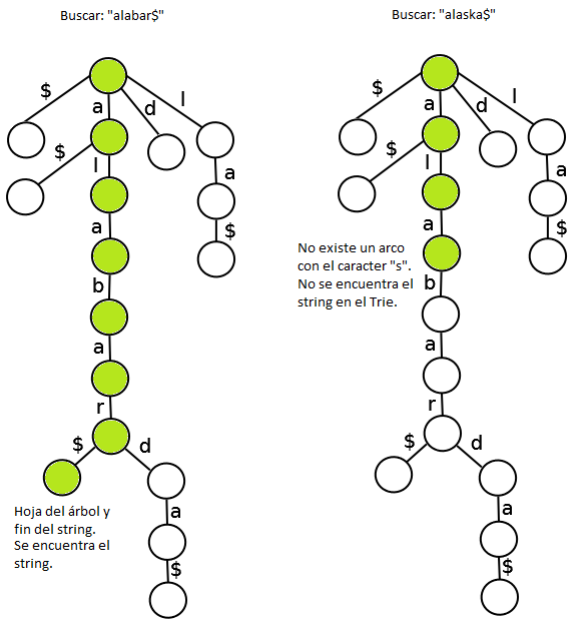


Para buscar un string en un Trie se empieza en la raíz del árbol, buscando un subárbol hijo cuyo arco que los conecta contenga el primer caracter del string. Si éste no existe, entonces el string buscado no está almacenado en el Trie. De lo contrario se procede de forma recursiva, esta vez con la raíz del subárbol y el segundo caracter del string (y así sucesivamente).

La búsqueda termina cuando se llega al último caracter del string. Si el último nodo visitado es una hoja, entonces el string se encuentra en el Trie, de lo contrario el string no se encuentra.

Ejemplo: Buscar "alabar$" y "alaska$" en el Trie del ejemplo anterior.

[1] El nombre Trie procede de information reTRIEval.



¿Qué nos proporciona un recorrido in-order de las hojas?

- Strings en orden lexicográfico.

¿Complejidad de búsqueda de un string de largo m?

- $O(m \times access\_child)$
- Por tanto, depende de cómo se soporte la operación *access_child* en la implementación concreta. Las más populares son:

	Espacio	Tiempo
List	O(T)	O(mσ)
Vector	O(T$\sum$)	O(m)
BST	O(T)	O(m log$\sum$)
Hash	O(T)	O(m) w.h.p.

Donde T es el número de nodos en el árbol, $\sum$ es el número de caracteres en el alfabeto.

Otras implementaciones (para soportar predecesor/sucesor):

	Espacio	Tiempo
vEB/y-fast	O(T)	O(mloglog$\sum$)
Suffix trays	O(T)	O(m+log$\sum$)

En este proyecto se puede optar por una implementación basada en Vector o por una basada en List. En cualquier caso, se debe argumentar bajo qué circunstancias (descripción del dominio) esa sería la mejor opción.