Curso 2018-2019
Estructura de Datos y Algoritmia
Práctica 3

Departamento de Lenguajes y Sistemas Informáticos

Objetivo

En esta práctica trabajaremos con **tipos de datos no lineales** del API de Java, construídos por el usuario y, además, usaremos algunos esquemas algorítmicos.

Fechas importantes

- Plazo de entrega de la tercera práctica (convocatoria ordinaria): desde el lunes 17 de diciembre hasta el **viernes 21 de diciembre de 2018**.

Más información al final de este documento.

1. Clase TIndice

La clase **TIndice** que hay que implementar contendrá

- las siguientes variables de instancia básicas (se podrán añadir las que se consideren necesarias justificando su inclusión):
  - `private TreeMap<String, TreeSet<Integer>> ti;`
- y los siguientes métodos de instancia básicos (se podrán añadir los que se consideren necesarios justificando su inclusión):
  - `public TIndice()`: inicializa las variables de instancia.
  - `public boolean esVacio()`: nos indica si el índice está vacío.
  - `public void insertaColeccion( Coleccion c )`: Si el índice no está vacío cuando se invoca al método pueden ocurrir dos cosas:
    - si la palabra ya existe en el índice, hay que añadir al nodo correspondiente los documentos de la nueva colección en los que aparece (sin repetición de identificadores de documentos);
    - si la palabra no estaba en el índice, se añade un nuevo nodo al árbol con la palabra y todos los documentos de la colección en los que aparece.

- `public TreeSet<Integer> borra( String p )`: elimina del árbol el nodo que contiene la palabra que coincide (ignorando mayúsculas y minúsculas) con la pasada como parámetro, devolviendo el `TreeSet` de identificadores de documentos asociados a esa cadena.
- `public Set<String> getPalabras()`: devuelve el conjunto de palabras clave contenidas en el `TreeMap`.
- `public TreeSet<Integer> getDocum( String p )`: devuelve el `TreeSet` con los índices de documentos en los que aparece la palabra (ignorando mayúsculas y minúsculas) pasada por parámetro si la encuentra en el índice. En cualquier otro caso devuelve una referencia vacía (`null`).
- `public boolean inserta( String p )`: si la cadena no está en el índice, inserta un nodo al árbol con la cadena en minúsculas (con un `TreeSet` vacío como valor asociado) devolviendo `true`. Si la cadena ya está en el índice devuelve `false`;
- `public boolean agregaId( String p, int i )`: añade un nuevo identificador de documento al vector asociado al nodo etiquetado con la cadena que coincide con la cadena pasada como parámetro ignorando minúsculas y mayúsculas; devuelve `true` si lo agrega y `false` en cualquier otro caso, teniendo en cuenta que no puede haber identificadores repetidos en el vector;
- `public void escribeIndice()`: escribe el índice en la salida estándar, con el siguiente formato (tener en cuenta que los identificadores de los documentos deben estar ordenados, según este ejemplo):

```
agenda * 1 - 2 - 5 - 8 - 12 - 13 - 15 - 20 - 22
deporte * 1 - 3 - 4 - 6 - 8 - 13 - 20
guerra *
hombre * 2 - 3 - 4 - 6 - 7 - 9 - 21
negocio * 1 - 2 - 7 - 11 - 13 - 15
tiempo * 1 - 2 - 6
...
ley * 2 - 3 - 14 - 17 - 22
```

Vamos a realizar consultas similares a las realizadas en la práctica 2.

2. Uso de índices invertidos

La estructura antes implementada nos va a servir como índice invertido de documentos, es decir, dado un diccionario de palabras y una colección de documentos, nos indica para cada palabra almacenada en el diccionario los documentos en los que

aparece. Con esta información vamos a realizar algunos tipos de consulta.

Los tipos de consulta se realizarán con las siguientes operaciones:

- Unión (OR), representada en la aplicación por `O`
- Intersección (AND), representada en la aplicación por `A`

Implementa una clase denominada `ConsultaIndice2` que recibirá como parámetros de entrada cinco argumentos:

- `arg1` representa el diccionario
- `arg2` representa la colección de documentos
- `arg3` representa la primera cadena de la consulta
- `arg4` representa el operador (`A, O`)
- `arg4` representa la segunda cadena de la consulta

Ejemplos de consultas:

- `ConsultaIndice2 arg1 arg2 hombre A mujer`
- `ConsultaIndice2 arg1 arg2 hombre O mujer`

En la clase `ConsultaIndice2` se ejecutará la aplicación, y se implementará un método `main` que:

1. detecte los 5 parámetros de la aplicación (**siempre serán 5**);
2. abra los ficheros de texto con el diccionario y los documentos y los almacene en el tipo de datos que se han implementado en esta práctica (el `TIndice`);
3. realice una consulta concreta, mostrando los identificadores de los documentos que respondan a la consulta. Por ejemplo:

   ```
   java ConsultaIndice2 Dic1.txt Doc1.txt hombre O mujer
   SOL = 1 - 3 - 10
   ```

   Si no encuentra ningún documento que responda a la consulta la salida será:

   ```
   SOL = No existe
   ```

4. **Se mostrará el índice creado (independientemente de que haya solución o no).**

3. Corrección de errores tipográficos

A veces ocurre que los textos que queremos clasificar contienen errores tipográficos. En ese caso, es más difícil realizar las tareas de recuperación de información, ya que las palabras no coinciden exactamente con las almacenadas en el diccionario. Para resolver este problema, vamos a realizar previamente a cualquier tarea de recuperación de información otra que realizará una corrección de este tipo de errores.

Una forma típica de corregir estos errores es, dada una palabra, encontrar la palabra más parecida en un diccionario (o un número de ellas). Entonces, se trata de sustituir la palabra errónea por la más parecida.

3.1 Distancia de edición

Dadas dos cadenas s_1 y s_2 (de longitud n y m), la distancia de edición entre ellas es el número mínimo de *operaciones de edición* necesarias para transformar una en la otra. Las operaciones de edición permitidas son:

- insertar un carácter en la cadena
- borrar un carácter de la cadena
- sustituir un carácter de la cadena por otro

El algoritmo que se aplica para calcular la distancia de edición entre dos cadenas está basado en un esquema de *programación dinámica*. Por lo tanto se utiliza una matriz de tamaño $(n+1)*(m+1)$ para almacenar los resultados parciales que se van obteniendo y el resultado final.

ALGORITMO:

1. Se construye una matriz `D` con $m+1$ filas y $n+1$ columnas. Se inicializa la primer fila de la matriz con la secuencia 0, 1, 2, ..., n y la primera columna de la matriz con la secuencia 0, 1, 2, ..., m;
2. se coloca cada carácter de la cadena s_1 en su correspondiente celda j (j va de 1 a n);
3. se coloca cada carácter de la cadena s_2 en su correspondiente celda i (i va de 1 a m);
4. los costes asociados a cada operación se definen como:
   - coste de inserción $c_i = 1$; [1]
   - coste de borrado $c_b = 1$; [2]

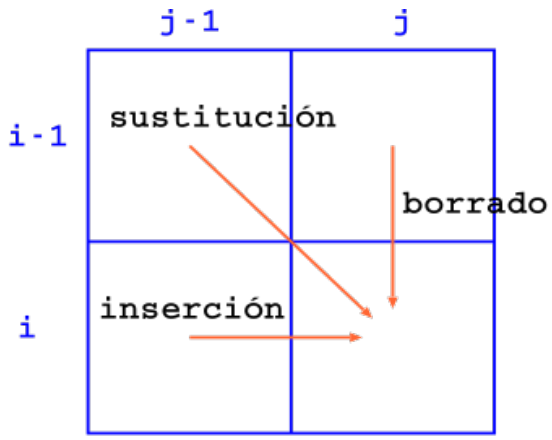
[1] se suele expresar también como $c(\lambda, x_i)$, que significa sustituir λ por x_i (siendo λ la cadena vacía).

[2] se suele expresar también como $c(x_i, \lambda)$, que significa sustituir x_i por λ (cadena vacía).

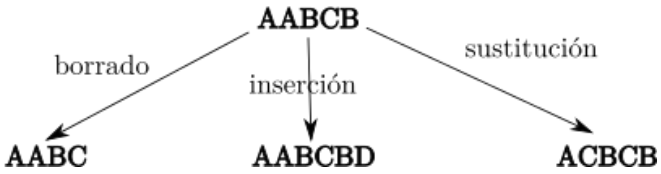
- coste de sustitución c_s:

$$\texttt{coste de sustitución}(c_s) = \begin{cases} 1 & \texttt{si } s_1(j) \neq s_2(i) \\ 0 & \texttt{si } s_1(j) = s_2(i) \end{cases}$$

5. El valor de la celda $\texttt{D}(i, j)$ es el mínimo de:
   - Valor de la celda $(i-1, j) + c_b$ (`borrado`)
   - Valor de la celda $(i, j-1) + c_i$ (`inserción`)
   - Valor de la celda $(i-1, j-1) + c_s$ (`sustitución`)
6. La distancia es la celda $\texttt{D}(n, m)$



Ejemplo de transformación de las cadenas:



Ejemplo de distancia de edición entre las cadenas *aab* y *abd*:

D	λ	a	a	b
λ				
a				
b				
d				

D	λ	a	a	b
λ	0	1	2	3
a	1			
b	2			
d	3			

D	λ	a	a	b
λ	0	1	2	3
a	1	0	1	2
b	2			
d	3			

D	λ	a	a	b
λ	0	1	2	3
a	1	0	1	2
b	2	1	1	1
d	3	2	2	2

Para evitar el efecto de la longitud de las cadenas en la medida de distancia [3], muchas veces la distancia de edición se suele normalizar. En nuestro caso el tipo de normalización que aplicaremos es dividir la distancia obtenida por la suma de las longitudes de las dos cadenas, es decir:

$$D_{normalizada} = \frac{D(n,m)}{n+m}$$

3.2 Aplicación

Implementa una clase denominada `CorrigeErrores` que recibirá como parámetros de entrada tres argumentos:

- `arg1` representa el diccionario
- `arg2` representa la colección de documentos a corregir (normalmente uno)
- `arg3` representa (`S, N`) donde
  - `S`: resultados con la distancia sin normalizar
  - `N`: resultados con la distancia normalizada

[3] no es lo mismo un error en la distancia entre dos cadenas de longitud 2 que en la distancia entre dos cadenas de longitud 20

En la clase `CorrigeErrores` se ejecutará la aplicación, que consiste en:

1. para cada palabra de los `Token` del vector `general` de los documentos, encuentra la palabra más parecida [4] en el diccionario utilizando la distancia de edición. Pueden pasar dos cosas:
   - si la distancia es cero, es que se trata de la misma cadena, y por lo tanto la palabra del documento está en el diccionario y está correctamente escrita;
   - si la distancia es distinta de cero significa que existe una muy parecida con mínimos cambios (normalmente debido a un error tipográfico);
2. se trata de mostrar por la salida estándar las palabras de los documentos con errores (aquellas que se encuentran a distancia distinta de cero), la palabra propuesta como corrección y la distancia a la que se encuentra. Si la distancia está normalizada, el resultado se mostrará con una precisión de 4 dígitos decimales [5] (ver Apéndice para obtener este formato).

Se implementará un método `main` que:

- detecte los parámetros de la aplicación;
- abra los ficheros de texto con el diccionario y los documentos y lo almacene todo utilizando los tipos de datos adecuados [6];
- muestre por pantalla las palabras erróneas, una por línea, según se ha explicado en el párrafo anterior.

Por ejemplo:

- `java CorrigeErrores arg1 arg2 N`

```
PALABRAS ERRONEAS CON DISTANCIA NORMALIZADA
aqua agua 0,125
meprte deporte 0,1538
ley li 0,4
tempo tiempo 0,0909
```

[4] significa encontrar la palabra que se encuentra a menor distancia (si hay más de una, la primera que se encuentra)

[5] IMPORTANTE: cuando se busca la palabra más cercana hay que normalizar la distancia antes de buscar el mínimo.

[6] no es necesario el `TIndice`

- `java CorrigeErrores arg1 arg2 S`

  ```
  PALABRAS ERRONEAS CON DISTANCIA SIN NORMALIZAR
  aqua agua 1
  meprte deporte 2
  ley li 2
  tempo tiempo 1
  ```

4. Normas generales

Entrega de la práctica:

- **Lugar de entrega:** servidor de prácticas del DLSI, dirección `http://pracdlsi.dlsi.ua.es`;
- **Plazo de entrega:** desde el lunes 17 de diciembre hasta el **viernes 21 de diciembre a las 23:59 horas**;
- Se deben entregar las prácticas en **un fichero comprimido**, con todos los ficheros creados y ningún directorio de la siguiente manera:
  - `tar cvfz practica3.tgz Token.java Docum.java Coleccion.java TIndice.java ConsultaIndice2.java CorrigeErrores.java`
- No se admitirán entregas de prácticas por otros medios que no sean a través del servidor de prácticas;
- El usuario y contraseña para entregar prácticas es el mismo que se utiliza en UACloud;
- La práctica se puede entregar varias veces, pero sólo se corregirá la última entregada;
- Los programas deben poder ser compilados sin errores ni warnings con el compilador de Java existente en la distribución de Linux de los laboratorios de prácticas; si la práctica no se puede compilar su calificación será 0. Se recomienda compilar y probar la práctica con el autocorrector inmediatamente antes de entregarla;
- Los ficheros fuente deber estar adecuadamente documentados usando comentarios en el propio código, sin utilizar en ningún caso acentos ni caracteres especiales;
- Es imprescindible que se respeten estrictamente los formatos de salida indicados, ya que la corrección principal se realizará de forma automática;

- Al principio de cada fichero entregado (primera línea) debe aparecer el DNI y nombre del autor de la práctica (dentro de un comentario) tal como aparece en UACloud (pero sin acentos ni caracteres especiales);

  Ejemplo:
  `DNI 23433224 MUÑOZ PICÓ, ANDRÉS` ⇒ NO
  `DNI 23433224 MUNOZ PICO, ANDRES` ⇒ SI

Sobre la evaluación en general:

- La práctica debe ser un trabajo original de la persona que entrega; en caso de detectarse indicios de copia de una o más entregas se suspenderá la práctica con un 0 a todos los implicados;
- La influencia de la nota de esta práctica sobre la nota final de la asignatura está publicada en la ficha oficial de la asignatura (apartado evaluación).

7. Probar la práctica

- En UACloud se publicará un corrector de la práctica con algunas pruebas (se recomienda realizar pruebas más exhaustivas).
- El corrector viene en un archivo comprimido llamado `correctorP3.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar: `tar xfvz correctorP3.tgz`.

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica3-prueba`: dentro de este directorio están los ficheros
    - `p01.java`: programa fuente en Java con un método main que realiza una serie de pruebas sobre la práctica.
    - `p01.txt`: fichero de texto con la salida correcta a las pruebas realizadas en `p01.*`.
- Una vez que tenemos esto, debemos copiar nuestros ficheros fuente (TODOS) al mismo directorio donde está el fichero `corrige.sh`.
- Sólo nos queda ejecutar el *shell-script*. Primero debemos darle permisos de ejecución. Para ello ejecutar:

  ```
  chmod +x corrige.sh
  corrige.sh
  ```

APENDICE: Escritura de números reales con formato en Java

- En primer lugar debemos indicarle a Java el local (idioma nativo) con el que vamos a trabajar, ya que cada idioma tiene sus características propias a la hora de escribir los números [7]. En programación los locales se identifican por cadenas que los representan. Por ejemplo, en linux:

  - `es_ES.utf8` → español de España, codificación en utf8.
  - `en_GB.utf8` → inglés del Reino Unido, codificación utf8.
  - `de_DE.ISO8859-1` → alemán de Alemania, codificación Latin1

  donde el primer par de caracteres indican el idioma, el segundo (separados por un subrayado) el país y por último se indica la codificación.

  Para indicarle a Java el locale con el que vamos a trabajar tenemos que crear un objeto de tipo Locale pasándole como parámetro el idioma. Por ejemplo, para indicarle que vamos a trabajar en inglés sería:

  ```
  Locale idioma=new Locale(‘‘en’’);
  ```

- A continuación necesitamos indicar que vamos a trabajar con números reales con formato y tenemos que fijar su locale. Para ello, creamos un objeto de tipo DecimalFormatSymbols pasándole como parámetro el locale anterior, de manera que fijamos las características propias de este locale para escribir números (separador de decimales, miles, porcentaje, etc.). Por ejemplo:

  ```
  DecimalFormatSymbols caracs=new DecimalFormatSymbols(idioma);
  ```

- Por último creamos un objeto de tipo DecimalFormat al que le especificaremos el número de decimales que queremos mostrar y el formato para los números que hemos creado antes. Para obtener el número formateado hay que llamar a su método `format` pasándole como parámetro el número, que devuelve como un `String`. Por ejemplo,

```
DecimalFormat numero = new DecimalFormat("#.####",caracs);
double ejemplo1 = 34.97811;
double ejemplo2 = 34.978;
System.out.println(numero.format(ejemplo1));
                                        //en pantalla: 34.9781
System.out.println(numero.format(ejemplo2));
                                        //en pantalla: 34.978
```

- Para la práctica utilizaremos como idioma el inglés ya que la mayoría de bases de datos están escritas utilizando esta codificación.

[7] Por ejemplo en español los miles se separan con puntos y los decimales con comas, mientras que en inglés es exactamente al revés.