Curso 2023-2024
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 18 de diciembre hasta el **viernes 22 de diciembre de 2023**.

Más información al final de este documento.

1. Clases

1.1. Clase Card

La clase **Card** tendrá las siguientes variables de instancia:

- `private String word: cadena que contiene la palabra;`
- `private int frequency: frecuencia con la que aparece la palabra.`

Y los siguientes métodos:

- `public Card( String )`: constructor al que se le pasa por parámetro una cadena que debe almacenar en word en minúsculas e inicializar a 1 la frecuencia [1].
- `public boolean equals( Card )`: devuelve cierto cuando la palabra (*word*) del objeto pasado por parámetro es igual a la del propio objeto sin tener en cuenta su frecuencia. En cualquier otro caso devuelve falso.
- `public void increment()`: incrementa en una unidad la frecuencia.
- `public int getFrequency()`: devuelve la frecuencia.
- `public String getWord()`: devuelve la palabra.
- `public String toString()`: devuelve una cadena con la concatenación de la palabra, un espacio en blanco y la frecuencia de dicha palabra.

[1] si el parámetro es `null` o la cadena vacía, la frecuencia será 0 y en word se almacena la cadena vacía

1.2. Clase Document

La clase **Document** que hay que implementar tendrá las siguientes variables de instancia:

- `private int id`: el identificador del documento;
- `private String category`: cadena que indica la categoría del documento;
- `private ArrayList<Card> general`: contendrá las palabras que aparecen en un documento con su frecuencia.

Y los siguientes métodos:

- `public Document( int,String )`: constructor al que se le pasa un entero para el identificador y una cadena para la categoría.
- `public Card moreFrequently()`: devuelve la card cuya frecuencia sea mayor en el array general. Si hay más de uno devuelve el primero que se ha encontrado. Si el array general está vacío se devuelve `null`.
- `public void addCard( String )`: se le pasa por parámetro una cadena de manera que agrega una nueva card al final del array general si no existe ya la cadena. Si ya existe se incrementará su frecuencia.
- `public int getId()`: devuelve el identificador del documento.
- `public String getCategory()`: devuelve la categoría del documento.
- `public boolean search( String )`: devuelve cierto si la cadena pasada como parámetro coincide, ignorando diferencias entre mayúsculas y minúsculas, con alguna palabra de las cards contenidas en el array general del documento.
- `public ArrayList<Card> getGeneral()`: devuelve el array `general`.
- `public String toString()`: devuelve una cadena con la concatenación del identificador, un guión (-), la categoría, nueva línea y la información de las cards del array general una por línea (con el formato especificado en su método `toString`).

1.3 Clase Compendium

La clase **Compendium** que hay que implementar tendrá las siguientes variables de instancia:

- `private ArrayList<Document> documents`: array dinámico de objetos de tipo `Document`, para almacenar los textos leídos desde fichero;
- `private ArrayList<String> dictionary`: array dinámico de cadenas, para almacenar el diccionario leído desde fichero.

Y los siguientes métodos:

- `public Compendium()`: constructor por defecto que inicializa las variables de instancia.

- `public String toString()`: devuelve la cadena con la concatenación de la cadena "`DICTIONARY:`", cambio de línea, el contenido del array `dictionary` con las cadenas separadas por un espacio en blanco, cambio de línea y los documentos del array `documents` con el formato indicado en el método `toString()` de la clase `Document`.

- `public void readFile( String )`: se le pasa por parámetro una cadena que se corresponde con el nombre de un fichero. El método no propaga excepciones, por tanto deben ser tratadas en el propio método. Su tratamiento consistirá en mostrarlas por pantalla[2]. El método lee el fichero de texto con el formato descrito en la sección 1.1 de la práctica 1 de manera que:

  - si encuentra una etiqueta <DOC> se creará un objeto de tipo `Document` que se guardará al final de su array `documents`. Cada nuevo objeto se crea pasándole:
    - su identificador (que se corresponde con su posición en el fichero de texto empezando en 1)
    - su categoría (primera línea que aparece después de la etiqueta <DOC>
    - el texto (líneas sucesivas posteriores a la anterior). Cada vez que se lee una palabra[3], se le pasa al objeto `Document` para que la agregue a su vector `general` de `Card`. La primera vez que aparece la palabra su frecuencia es 1, y si ya ha aparecido anteriormente en el documento sólo se incrementa su frecuencia en 1. Hay que tener en cuenta que la palabra se debe guardar siempre en minúscula.
  - si encuentra una etiqueta <DIC>, significa que comienza el diccionario por tanto, lee por líneas hasta el final del fichero, guardando cada línea en minúsculas al final del diccionario.

- `public ArrayList<Integer> search( String )`: devuelve un array con los identificadores de los documentos que contengan entre sus cards alguna palabra que coincida, ignorando diferencias entre mayúsculas y minúsculas, con la pasada por parámetro. Si ningún documento contiene la cadena pasada por parámetro devuelve `null`.

- `public ArrayList<Document> getDocuments()`: devuelve el array `documents`.

- `public ArrayList<String> getDictionary()`: devuelve el array `dictionary`.

1.4. Clase IndexTree

La clase **IndexTree** que hay que implementar contendrá la siguiente variable de instancia (se podrán añadir las que se consideren necesarias justificando su inclusión):

- `private TreeMap<String, TreeSet<Integer>> tree.`

[2]Se mostrarán por pantalla con `System.out.println()`

[3]Se consideran separadores de palabras los siguientes caracteres: `, : ; ? ! . ( )` y el espacio en blanco.

Y los siguientes métodos de instancia (se podrán añadir los que se consideren necesarios justificando su inclusión):

- `public IndexTree()`: inicializa la variable de instancia.
- `public boolean isEmpty()`: devuelve cierto si el árbol está vacío y falso en cualquier otro caso.
- `public boolean insert( String )`: si la cadena no es la cadena vacía, ni `null` y no está en el árbol, inserta un nodo en el árbol con la cadena en minúsculas (con un `TreeSet` vacío como valor asociado) devolviendo cierto. En cualquier otro caso devuelve falso.
- `public boolean insertId( String, int )`: añade un nuevo identificador de documento al árbol asociado al nodo etiquetado con la cadena que coincide[4] con la cadena pasada como parámetro; devuelve cierto si lo agrega y falso en cualquier otro caso (teniendo en cuenta que no puede haber identificadores repetidos).
- `public void insertCompendium( Compendium )`: se recorre el diccionario del compendium pasado por parámetro. Para cada palabra del diccionario pueden ocurrir dos cosas:
  - si la palabra ya existe en el árbol, hay que añadir al nodo correspondiente los documentos del objeto pasado por parámetro en los que aparece (sin repetición de identificadores de documentos);
  - si la palabra no estaba en el árbol, se añade un nuevo nodo al árbol con la palabra y todos los documentos del objeto pasado por parámetro en los que aparece (si no aparece en ningún documento, se añade con un valor asociado vacío).
- `public TreeSet<Integer> erase( String )`: elimina del árbol el nodo que contiene la palabra que coincide[5] con la pasada como parámetro, devolviendo el `TreeSet` de identificadores de documentos asociados a esa cadena. Si en el árbol no hay ningún nodo con esa cadena, el método devuelve una referencia vacía (`null`).
- `public ArrayList<String> erase( int )`: se le pasa por parámetro el identificador de un documento. Elimina de los valores asociados a las palabras del árbol dicho identificador. Devuelve un `ArrayList` con las palabras de las cuales se ha eliminado el identificador. Si en el árbol no aparece el identificador pasado por parámetro, el método devuelve una referencia vacía (`null`).
- `public TreeMap<String,TreeSet<Integer>> search( char )`: devuelve una copia del subconjunto del árbol[6] formado por las palabras que comiencen por el carácter pasado por parámetro. Si en el árbol no hay ningún nodo cuya cadena comience por el carácter pasado por parámetro el método devuelve una referencia vacía (`null`).

[4] Ignorando diferencias entre mayúsculas y minúsculas.
[5] Ignorando diferencias entre mayúsculas y minúsculas.
[6] Duplicando todos los elementos que forman este subárbol.

- `public TreeSet<String> search( int )`: se le pasa por parámetro el identificador de un documento. Devuelve un `TreeSet` con las palabras entre cuyos valores asociados se encuentre el identificador pasado por parámetro. Si en el árbol no aparece el identificador pasado por parámetro, el método devuelve una referencia vacía (`null`).

- `public Set<String> getWords()`: devuelve el conjunto de palabras clave contenidas en el `TreeMap`.

- `public TreeSet<Integer> getDocuments( String )`: devuelve el `TreeSet` con los identificadores de documentos en los que aparece la palabra[7] pasada por parámetro si la encuentra en el árbol. En cualquier otro caso devuelve una referencia vacía (`null`).

- `public String toString()`: devuelve el contenido del árbol en una cadena que debe tener el siguiente formato:

  - para cada nodo del árbol: clave del nodo, espacio en blanco, asterisco;
  - si el nodo tiene identificadores de documentos asociados: espacio en blanco, identificador. Si hay más de uno, para cada identificador, espacio en blanco, guión, identificador. Termina con cambio de línea sin espacio en blanco;

  de manera que al mostrar por pantalla la cadena que devuelve este método se vería como el siguiente ejemplo:

Terminal

```
war * 1 - 2 - 5 - 8 - 11 - 22
dog * 7 - 9 - 14 - 17
law *
house * 1 - 2 - 7 - 11 - 13
...
music * 22
```

2. Corrección de errores tipográficos

2.1. Distancia de edición

En algunas ocasiones la recuperación de información es más difícil ya que los textos pueden contener errores tipográficos, por tanto las palabras no coinciden exactamente con las buscadas. Para intentar minimizar este problema, se realizan tareas para la corrección de este tipo de errores.

[7] Ignorando diferencias entre mayúsculas y minúsculas.

La forma más habitual para corregir estos errores es encontrar la palabra más parecida que haya en un conjunto de palabras, de manera que se sustituye la palabra errónea por la más parecida.

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. Sean las cadenas s_1 y s_2 para las que se calcula la distancia de edición, donde n es la longitud de s_1 y m la longitud de s_2.

2. Se crea una matriz `D` de enteros con $n+1$ filas y $m+1$ columnas. Se inicializa la primera fila de la matriz con la secuencia 0, 1, 2, ..., m y la primera columna de la matriz con la secuencia 0, 1, 2, ..., n;

3. cada carácter de la cadena s_1 se corresponde con su fila i (i va de 1 a n);

4. cada carácter de la cadena s_2 se corresponde con su columna j (j va de 1 a m);

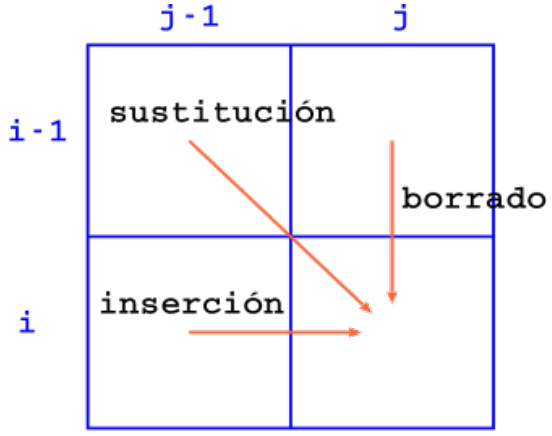
5. los costes asociados a cada operación se definen como:

   - coste de inserción $c_i = 1$;
   - coste de borrado $c_b = 1$;
   - coste de sustitución c_s:

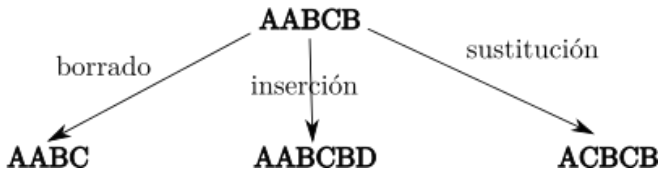
$$\texttt{coste de sustitución}(c_s) = \begin{cases} 1 & \texttt{si } s_1(i) \neq s_2(j) \\ 0 & \texttt{si } s_1(i) = s_2(j) \end{cases}$$

6. El valor de la celda `D`(i, j) es el mínimo de:

   - Valor de la celda $(i-1, j-1) + c_s$ (`sustitución`)
   - Valor de la celda $(i-1, j) + c_b$ (`borrado`)
   - Valor de la celda $(i, j-1) + c_i$ (`inserción`)



Ejemplo de transformación de las cadenas:



7. La distancia de edición entre s_1 y s_2 estará en la celda `D`(n, m)

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 [8], 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:

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

2.2 Aplicación

Implementa una clase `ErrorCorrector` con un `main` que recibirá como parámetros de entrada tres argumentos:

- `arg1`: opción para mostrar los datos normalizados o sin normalizar:
  - `W`: resultados con la distancia sin normalizar
  - `N`: resultados con la distancia normalizada
  - cualquier otro carácter o cadena es incorrecta
- `arg2`: nombre del fichero con los datos

[8] 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

- `arg3`: nombre del fichero con texto para corregir

La aplicación debe:

1. para cada palabra del fichero con texto para corregir se tiene que encontrar la palabra más parecida, lo que significa encontrar la palabra que se encuentra a menor distancia[9] (si hay más de una, la primera que se encuentra) almacenada en el árbol, ignorando diferencias entre mayúsculas y minúsculas, 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 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 del texto para corregir 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 (ver Apéndice para obtener este formato).

Para ello se implementará un método `main` que:

- detecte los parámetros de la aplicación. Si el número de argumentos es incorrecto o bien el argumento correspondiente a la opción de mostrar es incorrecta, se mostrará por pantalla el mensaje:

  Terminal
  ```
  Error: wrong arguments
  ```

  y no se realizará ninguna otra acción;
- abra el primer fichero de texto con la información y lo almacene todo utilizando el `IndexTree`;
- abra el segundo fichero de texto y procese cada palabra[10] tal y como se ha explicado en el párrafo anterior, mostrando por pantalla sólo las palabras erróneas (tal y como están escritas en el texto original), una por línea, según se ha explicado en el párrafo anterior. En cualquier caso se mostrará un mensaje indicando que opción se

[9] IMPORTANTE: cuando se busca la palabra más cercana con la opción de normalizar hay que normalizar la distancia antes de buscar el mínimo. Cuando se busca la palabra más cercana con la opción de no normalizar no hay que normalizar.

[10] Los separadores de palabras son: `, : ; ? ! . ( )` y el espacio en blanco

ha seleccionado (normalizado o sin normalizar) tal y como aparece en los ejemplos siguientes.

Por ejemplo:

Terminal

```
$ java ErrorCorrector N file1.txt file2.txt
WRONG WORDS (NORMALIZED)
Aqua agua 0.125
meprte deporte 0.1538
ley li 0.4
tempo tiempo 0.0909
```

Terminal

```
$ java ErrorCorrector W file1.txt file2.txt
WRONG WORDS (WITHOUT NORMALIZING)
Aqua agua 1
meprte deporte 2
ley li 2
tempo tiempo 1
```

3. 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 18 de diciembre hasta el **viernes 22 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 Card.java Document.java Compendium.java IndexTree.java ErrorCorrector.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 UA-Cloud;

- 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` $\Rightarrow$ NO
  `DNI 23433224 MUNOZ PICO, ANDRES` $\Rightarrow$ 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).

4. Probar la práctica

- En UACloud se publicará un corrector de la práctica con algunas pruebas (se recomienda realizar pruebas más exhaustivas).
- Podéis enviar vuestras pruebas (fichero con extensión java, con un método main y sin errores de compilación) a los profesores de la asignatura mediante tutorías de UACloud, para obtener la salida correcta a esa prueba **a partir del 11 de diciembre**. En ningún caso se modificará/corregirá el código de las pruebas. Los profesores contestarán a vuestra tutoría adjuntando la salida de vuestro main, si no da errores.
- 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-pruebas`: 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 [11]. 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, en nuestra práctica tenemos que indicarle que vamos a trabajar en inglés:

  ```
  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.):

  ```
  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.

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