Curso 2015-2016
Estructura de Datos y Algoritmia
Práctica 2 (12 de octubre)

Departamento de Lenguajes y Sistemas Informáticos

1. Objetivo

Con esta segunda práctica vamos a empezar a utilizar clases del `API Collections` y definiremos también nuestros propios tipos de datos lineales (listas). Seguiremos trabajando con la gestión de información relacionada con diccionarios utilizando diferentes estructuras de datos y algoritmos. En particular:

1. Uso del `API Collections`.
2. Implementación de estructuras de datos.
3. Aplicaciones concretas con las clases implementadas con diversos objetivos.

Plazo de entrega: desde el jueves 5 de noviembre hasta el **miércoles 11 de noviembre**. Más información al final de este documento.

1. Clases

Hay que implementar las siguientes clases (algunas de ellas nuevas y otras versiones de otras implementadas en la práctica 1), además de implementar un interface:

- `Palabra2` *//simplificación de* `Palabra` *con cambio en las variables de instancia*
- `DiccVector` *//diccionario que utiliza la clase Vector de java.util*
- `DiccLisJava` *//diccionario que utiliza una de las clases Lista de java.util*
- `DiccMiLista` *//diccionario que utiliza una Lista implementada en la práctica*
- `NodoL` *//nodos de una lista enlazada simple*
- `ListaBilingue` *//diccionario bilingüe que utiliza una Lista propia*

Podéis añadir los métodos que consideréis necesarios en cualquiera de las clases anteriores para la realización de la práctica, justificando en la documentación su necesidad.

No se puede usar el api de Java (métodos de ordenación) para ordenar ninguno de los diccionarios, se deben crear ordenados.

Los diccionarios a leer de fichero son del mismo tipo que los utilizados en la práctica 1, con la simplificación de que ahora no habrá acepciones.

Ahora, un fichero de texto para el diccionario podría ser el siguiente:

```
3
E F P
allow * permitir * permettre * permitir
at * a * chez * em
are * hay * sont * és
aunt * tía * tante *
after * después * après * depois
arrive * llegar * arriver * chegar
avenue * avenida * *
advertisement * publicidad * publicité * anúncio
something * algo * quelque chose * alguma coisa
everybody * todo el mundo * tout le monde * todo mundo
```

Como se puede observar en el ejemplo, es posible que el fichero no muestre la lengua origen (primera palabra) de forma ordenada, y además puede haber traducciones en alguna lengua que no existan. La cadena correspondiente a la lengua origen siempre existirá.

Clase Palabra2

La clase `Palabra2` que hay que implementar contendrá:

- las siguientes variables de instancia:
  - `private char[] lenguas;` //I inglés, E español, P portugués, F francés
  - `private String origen;` //palabra de la lengua de origen (inglés)
  - `private Vector<String> trad;` //almacenará una traducción de cada lengua
- y los siguientes métodos de instancia:

- `public Palabra2(String p, char[] lenguas)`: `p` es la palabra origen y `lenguas` es el array que contiene a qué idiomas traducimos. Si es `null`, por defecto será {'E','F','P'}. Los datos que no se pasan por parámetro se inicializan a sus valores por defecto;
- `public int setTrad(String t, char l)`: almacena la traducción `t` en la posición correspondiente al idioma en `trad` si dicha lengua no tiene ya una traducción distinta almacenada. Si ya hay una traducción anterior, se debe modificar para que sea la nueva. El método devuelve la posición de `trad` que ocupa la traducción almacenada o modificada, o -1 por defecto.
- `public String getOrigen()`: devuelve la cadena de la lengua origen del objeto;
- `public String getTraduccion(char l)`: devuelve una cadena que contiene la traducción en la lengua `l`. Por defecto el método devuelve `null`.
- `public void escribeInfo()`: muestra por pantalla las traducciones asociadas a la cadena origen; esta información se presentará en el mismo orden establecido en el diccionario leído de fichero de texto. El formato consiste en presentar la palabra origen seguida de ":", y a continuación cada nueva lengua (formada a su vez por cadenas o expresiones) con el mismo identificador.

  Por ejemplo, la salida para la palabra `''at''` en el ejemplo del enunciado será:

  ```
  at:a:chez:em
  ```

Interface Diccionario

Con `Palabra2` vamos a construir diferentes diccionarios con el mismo comportamiento pero diferentes estructuras de datos. Definiremos por tanto el interface `Diccionario` que implementarán estas clases. El interface `Diccionario` contiene los siguientes métodos de instancia:

- `public void leeDiccionario(String f)`: se leerá el diccionario desde un fichero de texto (que habrá que abrir y cerrar, y cuyo nombre le habremos pasado por parámetro). Se irán leyendo todas las líneas del fichero y se irá incorporando toda la información en las variables correspondientes definidas en cada clase. La información habrá que almacenarla de forma que el diccionario esté ordenado por la lengua origen. Este método no propaga excepciones, por lo tanto cualquier excepción que aparezca se tiene que tratar en el propio método, y dicho tratamiento será mostrar por pantalla el objeto `Exception`;

- `public boolean inserta(Palabra2 p)`: inserta una nueva palabra (con sus traducciones) en el diccionario (que estará ordenado). Si la palabra no traduce a las mismas lenguas y en el mismo orden que el diccionario la palabra no se inserta. En caso de que la cadena de la lengua origen ya exista comprueba las traducciones, añadiendo la traducción a la lengua correspondiente si no existía antes, o sustituyéndola en caso de que sí hubiese ya una traducción distinta a dicha lengua. El método devolverá `true` si realiza la inserción de una nueva palabra o traducción, y `false` en caso contrario; por ejemplo, supongamos que en el diccionario tenemos la siguiente palabra con sus traducciones:

  ```
  matter * materia * * questão
  ```

  y nos llega la siguiente palabra:

  ```
  matter * * affaire * assunto
  ```

  el resultado final que tiene que quedar es:

  ```
  matter * materia * affaire * assunto
  ```

  donde:

  - hemos mantenido la traducción de la primera lengua por no existir alternativa en la palabra a introducir
  - hemos añadido la traducción de la segunda lengua en la nueva palabara por no existir en la original
  - hemos sustituido la traducción de la tercera lengua por la nueva traducción, porque son distintas

- `public boolean borra(String s)`: borra la palabra cuya cadena de la lengua origen coincide con `s`. Si no encuentra la cadena devuelve `false`, en caso contrario `true`;

- `public int busca(String s)`: busca entre las cadenas de la lengua origen del diccionario la cadena `s` pasada como parámetro, y devuelve el número de comparaciones que realiza para encontrarla; si la cadena no existe devuelve el número de comparaciones en negativo; utiliza el mejor algoritmo de búsqueda en función de la estructura de datos;

- `public String traduce(String s, char l)`: busca entre las palabras de la lengua origen la cadena `s` pasada como parámetro y devuelve la traducción de la cadena en la lengua `l`; por defecto el método devuelve `null`;

- `public void visualiza()`: muestra todas las palabras del diccionario con todas las traducciones con el formato indicado para el método `escribeInfo` de la clase `Palabra2`;
- `public void visualiza(int j)`: muestra solo las j primeras líneas del diccionario con todas las traducciones con el formato indicado para el método `escribeInfo` de la clase `Palabra`;
- `public void visualiza(int j, char l)`: muestra solo las j primeras líneas del diccionario con la traducción de la lengua "l" (si no existe esa lengua en el diccionario muestra solo la lengua origen), por ejemplo, para `visualiza(4, 'P')` mostraría

```
allow:permitir
at:em
are:és
aunt:
```

Clase DiccVector

La clase `DiccVector` implementa la interfaz `Diccionario` y definirá las variables de instancia:

- `private int nlenguas`;
- `private Vector<Character> lenguas;` //E español, P portugués, F francés, por ejemplo, y pueden estar en cualquier orden
- `private Vector<Palabra2> dicc;` //almacena el diccionario ordenado por la lengua origen

y el siguiente método de instancia, además de la implementación de los definidos en el interface `Diccionario`:

- `public DiccVector()`: inicializa las variables de instancia a sus valores por defecto, y -1 para `nlenguas`;

Clase DiccLisJava

La clase `DiccLisJava` implementa la interfaz `Diccionario` y definirá las variables de instancia:

- `private int nlenguas`;

- `private Vector<Character> lenguas;` //E español, P portugués, F francés, por ejemplo, y pueden estar en cualquier orden
- `private XXX<Palabra2> dicc;` //almacena el diccionario ordenado por la lengua origen donde `XXX` podrá ser un `ArrayList` o un `LinkedList`.

y el siguiente método de instancia, además de los definidos en el interface `Diccionario`:

- `public DiccLisJava()`: inicializa las variables de instancia a sus valores por defecto, y -1 para `nlenguas`;

Clase DiccMiLista

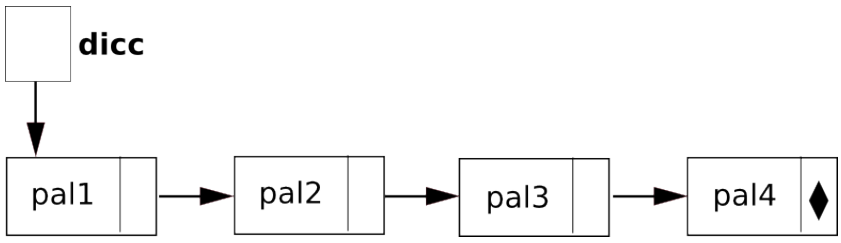
La clase `DiccMiLista` se definirá con las variables de instancia:

- `private int nlenguas`;
- `private Vector<Character> lenguas;` //E español, P portugués, F francés, por ejemplo, y pueden estar en cualquier orden
- `private NodoL dicc;` //almacena el diccionario ordenado por la lengua origen.

y el siguiente método de instancia, además de los definidos en el interface `Diccionario`:

- `public DiccMiLista()`: inicializa las variables de instancia a sus valores por defecto, y -1 para `nlenguas`;

Gráficamente:



Clase NodoL

Esta clase se definirá como privada de la clase `DiccMiLista` (de manera que sólo la lista puede acceder a un objeto de tipo `NodoL`). Contendrá

- las variables de instancia siguientes:
  - `private Palabra2 pal;`

  - `private NodoL next;`

- y los siguientes métodos de instancia:

  - `public NodoL()`: inicializa las variables de instancia a valores por defecto (null);
  - `public NodoL(Palabra2 p)`: inicializa el nodo con el objeto de tipo `Palabra2` pasado como parámetro, y el resto de variables de instancia con los valores por defecto (null);
  - `public void cambiaNext(NodoL n)`: cambia el valor del nodo apuntado por `next` a `n`;
  - `public void setPalabra2(Palabra2 p)`: cambia el valor de la variable de instancia `Palabra2` del nodo por `p`;
  - `public NodoL getNext()`: devuelve el nodo apuntado por `next`;
  - `public Palabra2 getPalabra2()`: devuelve la variable `Palabra2` contenida en el nodo;

3. Doble lista bilingüe

Vamos a diseñar una lista especial con un doble enlace pero con una interrelación entre los nodos diferente a la clásica que se establece entre los nodos de una lista doblemente enlazada. Esta lista especial almacenará un diccionario bilingüe que nos va a mantener ordenados en todo momento los dos diccionarios de las dos lenguas. Por lo tanto, nos permitirá en cualquier momento visualizar un listado ordenado o realizar una búsqueda ordenada por cualquiera de las dos lenguas con el mismo coste.

Vamos a ver con el siguiente ejemplo cómo se crearía la lista a partir del siguiente diccionario bilingüe inglés-español:

```
tail * cola
long * largo
is * es
king * rey
duck * pato
```

El diccionario ordenado por la primera lengua nos devolvería:

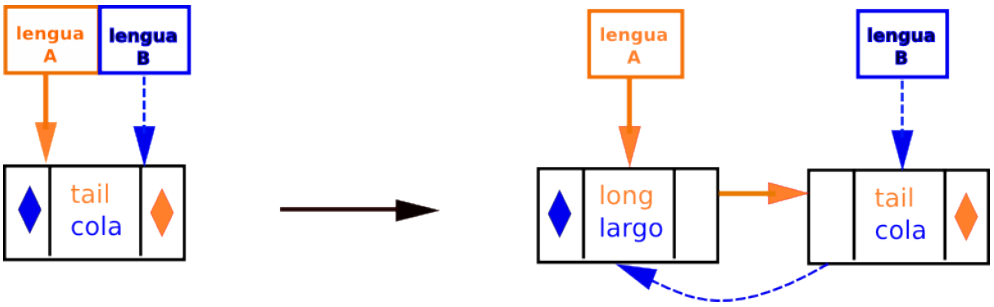
```
duck:pato
is:es
king:rey
long:largo
```

```
tail:cola
```

y ordenado por la segunda nos devolvería:

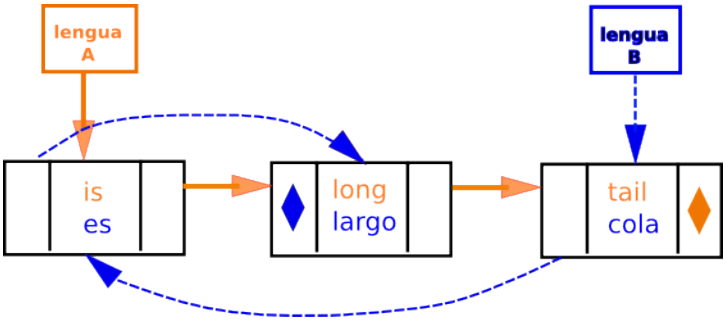
```
cola:tail
es:is
largo:long
pato:duck
rey:king
```

Empezamos insertando las parejas `tail-cola` y `long-largo`, creando un nuevo nodo para cada una de ellas. Como la "lengua A" está ordenada por la primera palabra del par, la pareja `long-largo` tenemos que ponerla como primer nodo de esta lengua, pero a la vez también como segundo nodo de la ‘‘`lengua B`’’. Es decir, en la lista ‘‘`lengua A`’’ es como si se realizara una inserción por la cabeza de la lista, y en la lista ‘‘`lengua B`’’ es como si realizáramos una inserción a la cola de la lista.

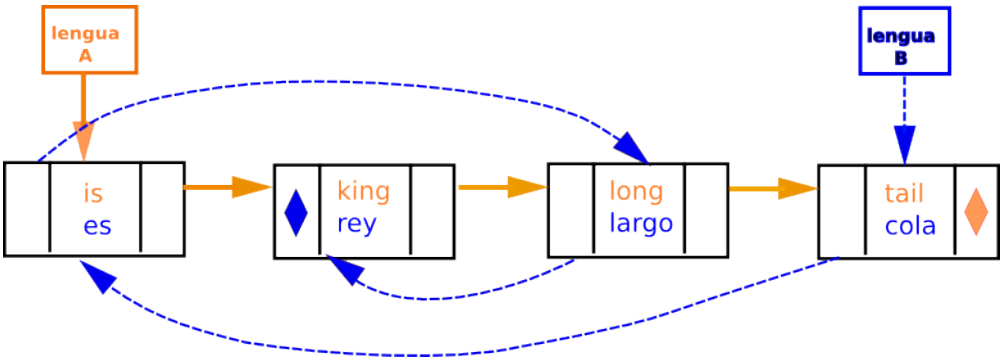


A continuación insertamos el par `is-es`, que pasará a ser primer nodo en la ‘‘`lengua A`’’ y un nodo intermedio en la ‘‘`lengua B`’’. Como se puede observar lo único que tenemos que hacer para conseguir realizar correctamente las operaciones es recorrer las listas en los dos sentidos, y actualizar de esta forma los enlaces de cada

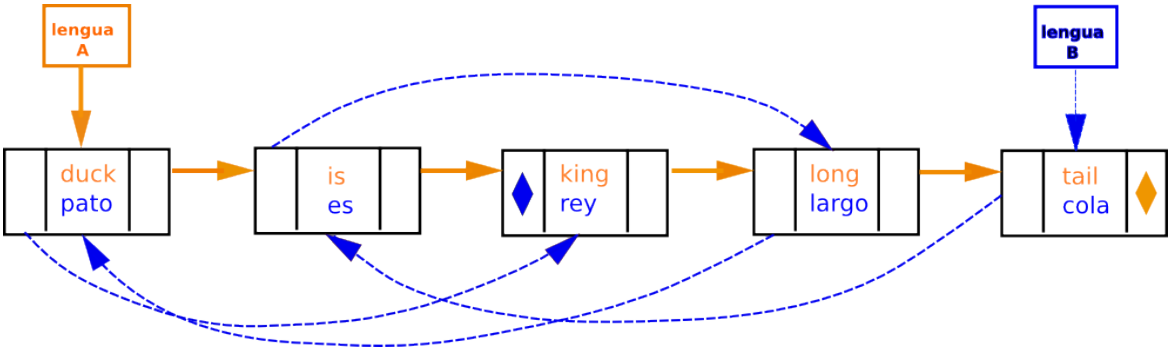
lista.



Ahora insertamos el par `king-rey`, que pasa a ser el segundo elemento en la ‘‘`lengua A`’’, y el último en la ‘‘`lengua B`’’:



Por último, insertamos el par `duck-pato`:



Se puede observar cómo se va modificando la lista y los enlaces entre los nodos de la lista para mantener ordenados los dos diccionarios. Si empezamos en el nodo ‘‘`lengua A`’’ y seguimos los enlaces azules (continuos) en el ejemplo estamos recorriendo de forma ordenada el diccionario inglés. Si empezamos el recorrido en el nodo ‘‘`lengua B`’’ y seguimos los enlaces naranjas (segmentados) estamos recorriendo de forma ordenada el diccionario español.

Se creará la clase `ListaBilingue` en la que tenéis que crear internamente una clase `NodoLD` con un doble enlace y que almacene las dos palabras de las dos lenguas. Además, definiremos al menos los siguientes métodos de instancia:

- `public ListaBilingue()`: inicializa las variables de instancia a sus valores por defecto;

- `public void leeDiccionario(String f)`: se leerá el diccionario desde un fichero de texto y se insertarán en la lista todas las palabras ordenadas según se ha explicado anteriormente en el ejemplo. Si alguna línea procesada no contiene las cadenas para las dos lenguas, no se inserta en la lista. Este método no propaga excepciones, por lo tanto cualquier excepción que aparezca se tiene que tratar en el propio método, y dicho tratamiento será mostrar por pantalla el objeto `Exception`;

- `public boolean inserta(String o, String d)`: inserta una nueva palabra (con su traducción) de forma ordenada en el diccionario. En caso de que la cadena de la lengua origen, ``o'' ya exista, o exista la cadena de la lengua destino, ``d'', no hay que insertar el par, ya que no se permite en el diccionario tener cadenas repetidas en ninguna de las dos lenguas; el método devolverá `true` si realiza la inserción, y `false` en caso contrario;

- `public boolean borraO(String s)`: borra la palabra del diccionario cuya cadena de la lengua origen coincide con `s`. Si no encuentra la cadena devuelve `false`, en caso contrario `true`;

- `public boolean borraD(String s)`: borra la palabra del diccionario cuya cadena de la lengua traducida coincide con `s`. Si no encuentra la cadena devuelve `false`, en caso contrario `true`;

- `public String buscaO(String s)`: busca entre las cadenas de la lengua origen la cadena `s` pasada como parámetro, y devuelve su traducción; si la cadena no existe devuelve null;

- `public String buscaD(String s)`: busca entre las cadenas de la lengua destino la cadena `s` pasada como parámetro, y devuelve su traducción; si la cadena no existe devuelve null;

- `public int indiceO(String s)`: busca entre las cadenas de la lengua origen la cadena `s` pasada como parámetro, y devuelve la posición en que se encuentra en la lista; si la cadena no existe devuelve -1; en el ejemplo anterior `IndiceO(``long'')` devolvería 4;

- `public int indiceD(String s)`: busca entre las cadenas de la lengua destino la cadena `s` pasada como parámetro, y devuelve la posición en que se encuentra en la lista; si la cadena no existe devuelve -1; en el ejemplo anterior `IndiceO(‘‘largo’’)` devolvería 3;

- `public void visualizaO()`: muestra todas las palabras del diccionario ordenado por la lengua origen, con todas las traducciones con el formato indicado para el método `escribeInfo` de la clase `Palabra2`;

- `public void visualizaD()`: muestra todas las palabras del diccionario ordenado por la lengua destino, con todas las traducciones con el formato indicado para el método `escribeInfo` de la clase `Palabra2`;

- `public Vector<String> getO(int i)`: devuelve el par de cadenas que se encuentra en la posición i de la lista origen; si dicha posición no existe devuelve `null`;

- `public Vector<String> getD(int i)`: devuelve el par de cadenas que se encuentra en la posición i de la lista destino; si dicha posición no existe devuelve `null`;

4. Aplicación: búsqueda de múltiples acepciones en las dos lenguas

Vamos a extender la funcionalidad de la clase `ListaBilingue` con una nueva forma de realizar las inserciones de los pares de cadenas de dos lenguas dadas, permitiendo añadir palabras que ya existan en el diccionario. Así podremos obtener un listado de todas las palabras de una lengua dada que tengan más de una traducción.

Para poder obtener este listado, deberemos definir un nuevo método en la clase `ListaBilingue` que permita insertar en la lista bilingüe palabras que ya existan en una de las dos lenguas. En concreto:

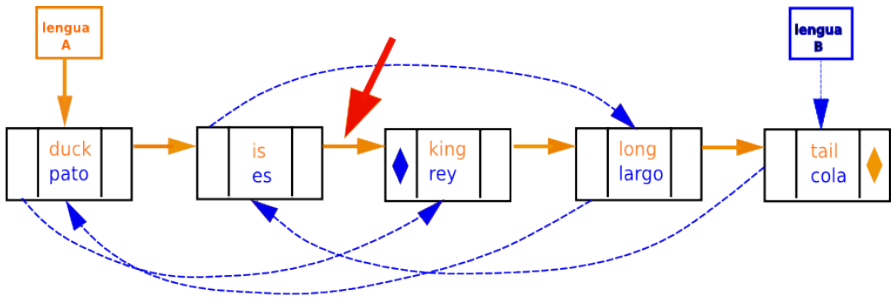
- si la palabra origen `‘‘pal1’’` ya existe en el diccionario (y no la traducción), crearemos un nuevo nodo que se insertará delante del nodo donde se encuentre la primera ocurrencia de dicha palabra;

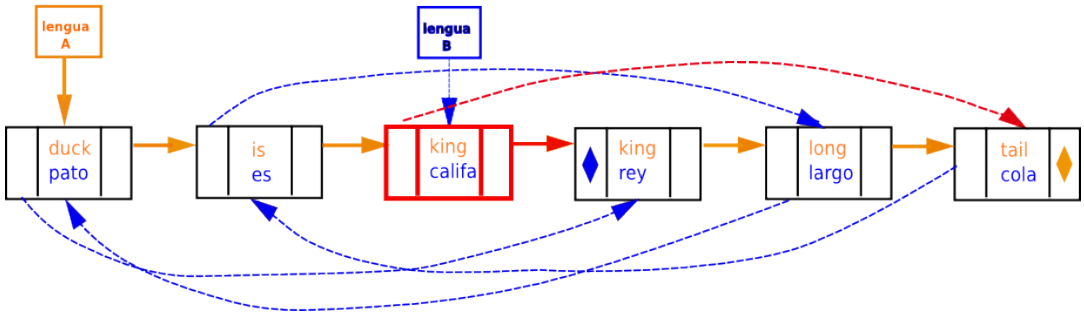
- si la palabra destino `‘‘pal2’’` ya existe en el diccionario (y no la palabra origen), crearemos un nuevo nodo que se insertará delante del nodo donde se encuentre la primera ocurrencia de dicha palabra;

- si existen las dos palabras, `‘‘pal1’’` y `‘‘pal2’’`, no se insertará la palabra;

Ejemplo: en el diccionario utilizado en la sección 3, y tras insertar las 5 palabras con sus traducciones, imaginemos que aparece a continuación la siguiente línea

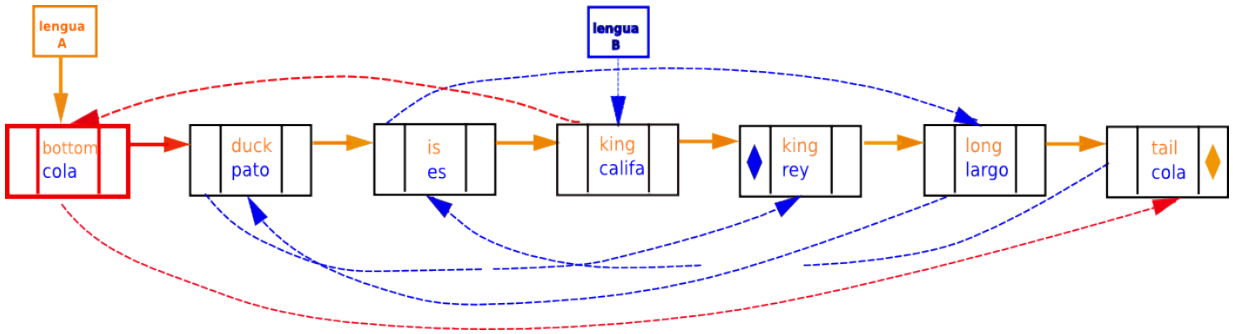
`king * califa`





y después la línea

`bottom * cola`



Implementa una clase denominada `ConsultaAcepciones` que recibirá como parámetros de entrada dos argumentos:

- `arg1` representa el diccionario
- `arg2` representa la lengua (O o D)

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

- detectará los parámetros de la aplicación;

- abrirá el fichero de texto con el diccionario y lo almacenará en una `ListaBilingue` según el método de inserción explicado en esta aplicación;
- se mostrará por pantalla un listado de todas las palabras de la lengua (O o D) con sus acepciones, en caso de tener más de una. Por ejemplo, en el caso de la última lista mostrada:

  ```
  java ConsultaAcepciones Dic.txt O
  ```

  ```
  SALIDA
  king:califa,rey
  ```

  ```
  java ConsultaAcepciones Dic.txt D
  ```

  ```
  SALIDA
  cola:bottom,tail
  ```

  Si no encontrara ninguna palabra con más de una acepción será:

  ```
  SALIDA
  No existe
  ```

Nota: aunque en estos ejemplos tan sencillos sólo ha aparecido una línea, es posible que aparezcan más, tantas como palabras en el diccionario tengan más de una acepción o traducción.

5. Documentación

Documenta en el mismo código (con un comentario antes de cada variable o método) **como mínimo** los siguientes elementos, con una breve descripción de cómo lo habéis hecho. Además contesta y justifica las preguntas que se hacen en cada clase, añadiendo dicha respuesta como comentario después de las variables de instancia y antes de los métodos de clase:

- las variables y métodos de instancia (o clase) añadidos por tí, justificando su necesidad;
- la aplicación;
- los siguientes métodos de clase:
  - de la clase `DiccVector`
    - `public boolean inserta(Palabra2 p)`: describe cómo has implementado este método (no cómo se define) y el coste total (asintótico) de todo el proceso;
    - `public int busca(String s)`: describe cómo has implementado este método (no cómo se define) y el coste total (asintótico) de todo el proceso;
  - de la clase `DiccLisJava`
    - `public boolean inserta(Palabra2 p)`: describe cómo has implementado este método (no cómo se define) y el coste total (asintótico) de todo el proceso;
    - `public int busca(String s)`: describe cómo has implementado este método (no cómo se define) y el coste total (asintótico) de todo el proceso;

    Nota: En ambas clases tenéis que tener en cuenta no sólamente el coste de vuestro código, sino el coste que conlleva el uso de la librería;
  - de la clase `ListaBilingue`
    - `public boolean inserta(String o, String d)`: describe cómo has implementado este método (no cómo se define) y el coste total de todo el proceso (además del asintótico);
    - `public int indiceD(String s)`: describe cómo has implementado (no cómo se define) este método y el coste total de todo el proceso (además del asintótico);
    - el método implementado necesario para la aplicación que permite insertar palabras que ya existen en alguna de las dos lenguas;

Normas generales

Entrega de la parte práctica:

- Lugar de entrega: servidor de prácticas del DLSI, dirección `http://pracdlsi.dlsi.ua.es`
- Plazo de entrega: desde el jueves 5 de noviembre hasta el **miércoles 11 de noviembre**.
- Se debe entregar la práctica en un fichero comprimido con todos los ficheros `.java` creados y ningún directorio de la siguiente manera

  ```
  tar cvfz practica2.tgz *.java
  ```

  En concreto los ficheros que hay que entregar son: `Diccionario.java DiccVector.java DiccLisJava.java Palabra2.java DiccMiLista.java ListaBilingue.java ConsultaAcepciones.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 el Campus Virtual.
- La práctica se puede entregar varias veces, pero sólo se corregirá la última entregada.
- Los programas deben poder ser compilados sin errores 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 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 el Campus Virtual (pero sin acentos ni caracteres especiales).

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

Sobre la evaluación en general:

- **El tiempo estimado por el corrector para ejecutar cada prueba debe ser suficiente para que finalice la ejecución de cada prueba correctamente, en otro caso se invoca un proceso que obliga a la finalización de la ejecución de esa prueba y se considera que dicha prueba no funciona correctamente.**
- 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 se detallan en las transparencias de presentación de la misma.

Probar la práctica

- En el Campus Virtual se publicará un corrector de la práctica con varias pruebas (se recomienda realizar pruebas más exhaustivas).
- El corrector viene en un archivo comprimido llamado `correctorP2.tgz`. Para descomprimirlo se debe copiar este archivo donde queramos realizar las pruebas de nuestra práctica y ejecutar: `tar xfvz correctorP2.tgz`.

  De esta manera se extraerán de este archivo:

  - El fichero `corrige.sh`: *shell-script* que tenéis que ejecutar.
  - El directorio `practica2-prueba`: dentro de este directorio están los ficheros
    - `p0X.dic`: fichero de texto con el diccionario usado en esta prueba;
    - `p0X.java`: programa en Java con un método main que realiza una serie de pruebas sobre la práctica.
    - `p0X.txt`: fichero de texto con la salida correcta a las pruebas realizadas en `p0X.java`.
- Una vez que tenemos esto, debemos copiar nuestros ficheros fuente (sólo aquellos con extensión .java) 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
```