Curso 2017-2018
Estructura de Datos y Algoritmia
Práctica 2 (versión 17 de octubre de 2017)

lsi Departamento de Lenguajes y Sistemas Informáticos

Objetivo

En esta práctica trabajaremos con tipos de datos del API de Java y construidos por el usuario. En particular:

- Uso del tipo `ArrayList` del API de Java con objetos de tipo `PLoc`.
- Implementación del tipo de datos lista de objetos de tipo `PLoc`.
- Aplicación de las estructuras de datos para resolver problemas de búsqueda.

Fechas importantes

Plazo de entrega de la segunda práctica: desde el lunes 13 de noviembre hasta el **viernes 17 de noviembre de 2017**.

1. Tipo Lista

Vamos a construir diferentes Listas para almacenar objetos de tipo `Ploc` con el mismo comportamiento pero diferentes estructuras de datos, de manera que es posible que métodos que realizan la misma acción tengan diferente implementación. Por ello definiremos el tipo `Lista` que contendrá los métodos comunes a todas las listas, y que puede ser una interfaz o una clase abstracta, de la que heredarán o implementarán las demás listas:

- `public void leeLista(String f)`: se leerá la lista 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, siempre por el principio. 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 esVacia()`: nos indica si la lista está vacía.

- `public void insertaCabeza(PLoc p)`: inserta un objeto de tipo `PLoc` al principio de la lista.
- `public void insertaCola(PLoc v)`: añade un objeto de tipo `PLoc` al final de la lista.
- `public void insertaArrayPLoc(PLoc[] v)`: añade al final de la lista los objetos de tipo `PLoc` que contiene el array pasado por parámetro.
- `public boolean borraCabeza()`: quita de la lista el primer objeto `PLoc` (a la cabeza de la lista), devolviendo `true` si no estaba vacía.
- `public boolean borraCola()`: quita el último objeto `PLoc` (a la cola) de la lista, devolviendo `true` si no estaba vacía.
- `public int ciudadEnLista(String v) throws CiudadNoEncontradaExcepcion`: devuelve la posición (empezando a contar desde 0) que ocupa el primer `PLoc` con la ciudad que coincide con `v`. Si la ciudad no está en ningún `PLoc` de la lista, lanza y propaga una excepción de tipo `CiudadNoEncontradaExcepcion` indicando el nombre de la ciudad no encontrada.
- `public boolean borraCiudad(String v)`: elimina el primer `PLoc` que contiene una ciudad en la lista que coincide con `v`, y devuelve `true`; si la ciudad que se pasa por parámetro no figura en ningún objeto `PLoc` de la lista, esta no se modifica y devuelve `false`.
- `public boolean borraPais(String s)`: quita de la lista todos los objetos de tipo `PLoc` que coinciden con el nombre de país pasado como parámetro y devuelve `true`; si el país que se pasa por parámetro no figura en la lista, esta no se modifica y devuelve `false`.
- `public PLoc getPLoc(int pos) throws IndexOutOfBoundsException`: devuelve el objeto `PLoc` contenido en la lista que ocupa la posición `pos` (empezando a contar desde 0). Si `pos` no es una posición válida de la lista el método debe lanzar y propagar la excepción `IndexOutOfBoundsException` indicando la posición no válida.
- `public PLoc[] Pais(String p)`: devuelve un array de `PLoc` con todas las ciudades del país pasado como parámetro (en el mismo orden que están en la lista). Si no hay ninguna se devuelve `null`.
- `public void ordenaLista()`: ordena de forma creciente la lista por la coordenada longitud decimal (en primer término), y en orden alfabético de la ciudad en segundo término de los objetos de tipo `PLoc` almacenados. Si hay dos objetos con la misma longitud y nombre de ciudad, el nuevo se almacenará detrás de los ya existentes.

2. Clase NodoLG

La clase **NodoLG** con la que construiremos la lista, que será una **clase privada** de la clase `ListaG`, contendrá

- las variables de instancia siguientes (tienes libertad para utilizar uno o dos objetos de tipo `NodoLG`):
  - `private PLoc pd;`
  - `private NodoLG next;`
- y los siguientes métodos de instancia:
  - `public NodoLG()`: inicializa las variables de instancia a valores por defecto (`null`).
  - `public NodoLG(PLoc p)`: inicializa el nodo con el objeto de tipo `PLoc` pasado como parámetro, y el resto de variables de instancia con los valores por defecto (`null`).

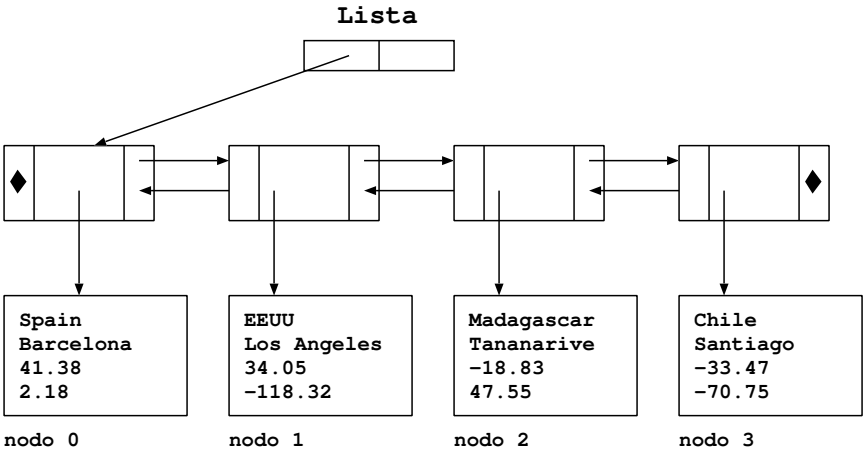
3. Clase ListaG

La clase **ListaG**, que tiene que implementar o heredar del tipo `Lista`, contendrá

- las variables de instancia siguientes (tienes libertad para utilizar uno o dos objetos de tipo `NodoLG`):
  - `private NodoLG pr;`
- y los siguientes métodos de instancia:
  - `public ListaG()`: inicializa las variables de instancia a sus valores por defecto (`null`).
  - `public void escribeListaG()`: escribe la lista, empezando desde la cabeza, a la salida estándar con el siguiente formato:

```
nodo 0: EU - Spain - Barcelona - 41.38 - 2.18
nodo 1: NA - EEUU - Los Angeles - 34.05 - 118.32
nodo 2: AF - Madagascar - Tananarive - 18.83 - 47.55
nodo 3: SA - Chile - Santiago - 33.47 - 70.75
```

Gráficamente (en el ejemplo no se está mostrando toda la información almacenada en el nodo, sólo una parte):



4. Clase VectorG

La clase `VectorG`, que tiene que implementar o heredar del tipo `Lista`, contendrá:

- la siguiente variable de instancia:
  - `private ArrayList<PLoc> vector;`
- y los siguientes métodos de instancia:
  - `public VectorG()`: crea un vector vacío (sin ningún elemento).
  - `public void escribeVectorG()`: escribe el vector, empezando desde la primera posición, a la salida estándar con el siguiente formato:

```
posic 0: EU - Spain - Barcelona - 41.38 - 2.18
posic 1: EEUU - Los Angeles - 34.05 - 118.32
posic 2: Madagascar - Tananarive - 18.83 - 47.55
posic 3: Chile - Santiago - 33.47 - 70.75
```

5. Implementación de otros métodos y clases.

Es necesario crear la clase `CiudadNoEncontradaExcepcion` que hereda de la clase `Exception`.

También es necesario implementar el método

```
public int compareTo( PLoc p );
```

en la clase `PLoc`. Este método es necesario para realizar la ordenación que se llevará a cabo según la longitud (en primer término) en orden creciente [1], y en orden alfabético de la ciudad en segundo término. El entero que se devuelve será:

- -1 si el valor de la longitud de la variable de instancia es menor que el valor de la longitud del objeto pasado como parámetro;
- 1 si el valor de la longitud de la variable de instancia es mayor que el valor de la longitud del objeto pasado como parámetro;
- si las longitudes son iguales devolverá:
  - -1 si la ciudad de la variable de instancia precede alfabéticamente a la ciudad del objeto pasado como parámetro;
  - 1 en caso contrario;
  - 0 si las dos ciudades son iguales (es decir, estamos en la situación en la que ambos objetos tienen la misma longitud y la misma ciudad).

6. Búsqueda de ciudades y regiones

Hoy en día nos podemos encontrar por la web muchos buscadores relacionados con localizaciones geográficas. Podemos buscar en *Google Maps* por país, ciudad o calle, o incluso por su latitud y longitud, tanto en coordenadas sexagesimales como decimales. En esta práctica vamos a realizar diferentes tipos de búsquedas simples.

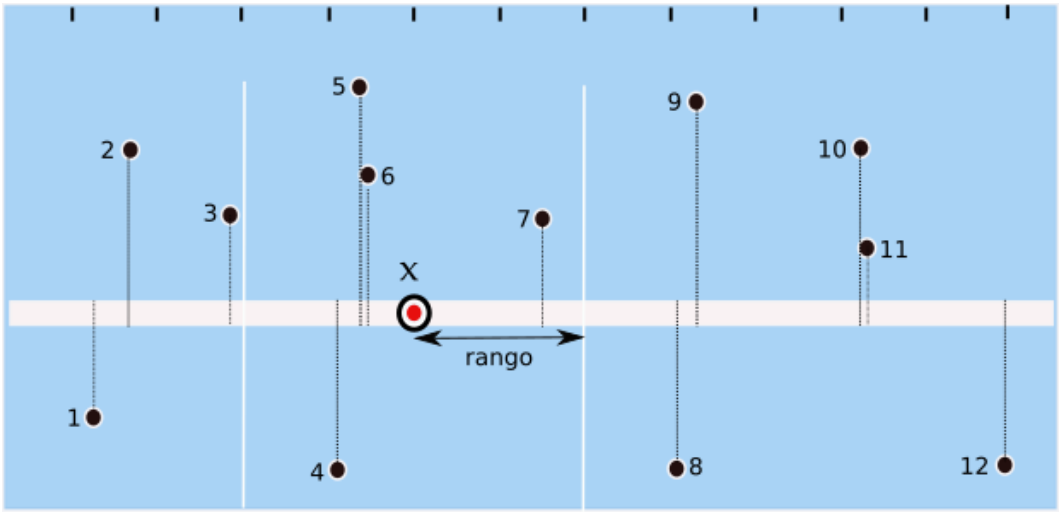
Implementa una clase denominada `BuscaLocalizacion` donde se ejecutará la aplicación. En la clase se implementará un método `main` que:

- detectará los parámetros de la aplicación, que tendrá 4 parámetros:
  - **Búsqueda por rango:**
    - nombre del fichero de la base de datos (string)
    - la letra 'R' (char)
    - longitud (double)
    - rango (double)
  - **Búsqueda por pares:**
    - nombre del fichero de la base de datos (string)
    - la letra 'P' (char)
    - nombre de un continente (string)
    - longitud máxima, Lmax (double)
- abra un fichero de texto con la base de datos de ciudades en formato sexagesimal, y lo almacene en el tipo de datos adecuado en función del tipo de búsqueda que se pida: la `ListaG` y el `VectorG`.

[1] la relación de orden para las coordenadas es la definida por su valor gps

- **Búsqueda por rango** [2]**.** Esta búsqueda debe ser realizada sobre el `VectorG`.

  - Si el cuarto parámetro (el rango) es 0, la aplicación debe invocar al método `escribeInfoGps()` de la ciudad cuya posición se acerque más a la longitud dada como parámetro (no se probarán los casos en los que haya 2 ciudades a la misma longitud). Si es negativo se mostrará únicamente el mensaje `EL VALOR DEL RANGO ES INCORRECTO`.
  - Si el rango es positivo, debe mostrar todos los nombres de ciudades (una por línea) que se encuentran dentro del rango (incluído el propio valor), ordenadas de menor a mayor por el valor absoluto de la diferencia de su longitud a la posición de referencia. Por ejemplo, si el paso de parámetros es: fichero.txt 23 2, hay que devolver los datos de todas las ciudades cuyo valor de la longitud en decimal esté entre 21 y 25, en orden creciente del valor absoluto de la diferencia de su longitud a la posición de referencia; a continuación se muestra un ejemplo gráfico donde si la búsqueda se pide en la posición dada por `x` con rango 2, la salida debe mostrar las localidades en las posiciones 4, 5, 6 y 7, y si el rango es 0 sólamente la localidad que ocupa la posición 6. En caso de que dentro del rango definido no haya ninguna localidad se mostrará el mensaje `NO HAY CIUDADES EN ESE RANGO`.



Ejemplo del formato de salida para el valor de rango 2 anterior:

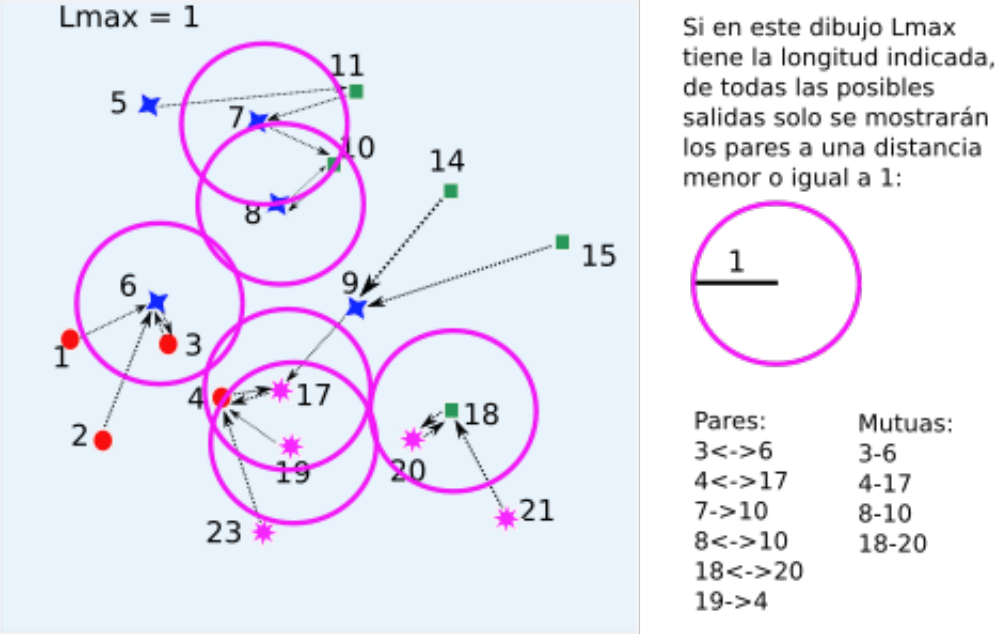
```
ciudad6
ciudad5
ciudad4
ciudad7
```

  - En este apartado se valorará la utilización del mejor método de búsqueda, y habrá qué describir qué método se ha elegido.

[2] en esta parte se trabajará sólo con datos unidimensionales, en concreto, sólo con la longitud

- **Búsqueda de ciudades fronterizas** [3]. Esta búsqueda se debe realizar sobre la `ListaG`.

  - Si `Lmax` es negativo o cero, debe mostrar el mensaje `EL VALOR DE LMAX ES INCORRECTO`.
  - Si `Lmax` es positivo, debe mostrar, de entre todas las ciudades de ese continente, aquella ciudad más cercana (aplicando la **distancia euclídea** utilizando las coordenadas decimales) del mismo continente pero de un país distinto que estén a una distancia menor o igual que `Lmax` (habrá que mostrar las dos ciudades implicadas separadas con un '-', cada par en una línea, ver ejemplo más abajo). El orden de presentación es de menor a mayor valor de los valores de distancia. Por ejemplo en la figura de abajo, la ciudad de otro país más cercana a 1 es 6, la del 2 es el 6, la del 3 es el 6, la del 4 es el 17, y así sucesivamente. Además, si se encuentra que existen pares que entre ellas son las más cercanas mutuamente, también se mostrará en la salida por separado. Por ejemplo, la pareja (4,17).
  - Si solo hay un país en ese continente, se mostrará el mensaje `NO HAY FRONTERAS INTRACONTINENTALES`.



  Ejemplo del formato de salida (ordenado por valor de la distancia, y de forma que la primera ciudad del par sea la que tiene más cercana a su pareja ($\rightarrow$) y si la pareja es mutua, por orden alfabético entre ellas dos):

```
CIUDADES FRONTERIZAS
ciudad3 (país) - ciudad6 (país)
ciudad18 (país) - ciudad20 (país)
```

[3] en esta parte se trabajará con datos bidimensionales, es decir, con la longitud y la latitud

```
ciudad4 (país) - ciudad17 (país)
ciudad19 (país) - ciudad4 (país)
ciudad8 (país) - ciudad10 (país)
ciudad7 (país) - ciudad10 (país)

CIUDADES FRONTERIZAS MUTUAS
ciudad3 (país) - ciudad6 (país)
ciudad18 (país) - ciudad20 (país)
ciudad4 (país) - ciudad17 (país)
ciudad8 (país) - ciudad10 (país)
```

El mensaje "CIUDADES FRONTERIZAS" siempre aparecerá en la ejecución, aunque no haya ninguna. Sin embargo, si no hay ciudades fronterizas mutuas, no debe salir el mensaje "CIUDADES FRONTERIZAS MUTUAS"

Documentación y cálculo de costes

Documenta en el mismo código (con un comentario antes de cada variable o método) **como mínimo** los siguientes elementos, con una breve descripción (o más extensa, dependiendo de las aclaraciones añadidas en cada método que se describe a continuación):

- las variables y métodos de instancia (o clase) añadidos por tí, justificando su necesidad e incluyendo la palabra "NEW" delante;
- la aplicación con todo detalle;
- los siguientes métodos de instancia:
  - de la clase `VectorG`:
    - `public void leeLista(String s)`
    - `public boolean borraPais(String s)`
    - `public PLoc[] Pais(String s)`
  - de la clase `ListaG`:
    - `public void leeLista(String s)`
    - `public boolean borraCiudad(String s)`
    - `public PLoc getPLoc(int i)`
    - `public void ordenaLista()`

Cálculo de los costes asintóticos

Obtén las cotas asintóticas (cota superior O) de los métodos siguientes, en el mismo orden que aparecen. Estos costes los muestras en el fichero de la aplicación. Hay que tener en cuenta que el tamaño del problema es el tamaño de la lista o el vector, y consideramos que en general es `n`.

Ejemplo:

```
COSTES ListaG:
1) int esVacia(): O(1)
2) void insertaCabeza(PLoc p):
3) void insertaCola(PLoc v):
4) boolean borraCabeza():
5) boolean borraCola():
6) int ciudadEnLista(String v):
7) boolean borraCiudad(String v):
8) boolean borraPais(String s):
9) PLoc getPLoc(int pos):

COSTES VectorG:
1) int esVacia(): O(1)
2) void insertaCabeza(PLoc p):
3) void insertaCola(PLoc v):
4) boolean borraCabeza():
5) boolean borraCola():
6) int ciudadEnLista(String v):
7) boolean borraCiudad(String v):
8) boolean borraPais(String s):
9) PLoc getPLoc(int pos):

COSTES aplicacion:
1) busqueda por rango:
2) busqueda por pares:
```

NOTA IMPORTANTE. Para escribir los costes hay que utilizar la siguiente notación:

- coste constante: `O(1)`
- coste logarítmico: `O(log(n)`
- coste lineal: `O(n)`
- cualquier otra combinación que requiera un producto, usar *, por ejemplo, un coste cuadrático sería `O(n*n)`, un coste lineal logarítmico sería `O(n*log(n))`

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 13 de noviembre hasta el **viernes 17 de noviembre**.
- Se debe entregar la práctica en un fichero comprimido los siguientes ficheros:
  - `PLoc.java`
  - `Coordenada.java`
  - `CoordenadaExcepcion.java`
  - `Lista.java`
  - `ListaG.java`
  - `VectorG.java`
  - `CiudadNoEncontradaExcepcion.java`
  - `BuscaLocalizacion.java`

  y ningún directorio de la siguiente manera

  `tar cvfz practica2.tgz PLoc.java Coordenada.java Lista.java ListaG.java CoordenadaExcepcion.java ListaG.java VectorG.java BuscaLocalizacion.java CiudadNoEncontradaExcepcion.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 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.
- 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
- Es imprescindible que se respeten estrictamente los formatos de salida indicados ya que la corrección se realizará de forma automática.
- Para evitar errores de compilación debido a codificación de caracteres que descuenta 0.5 de la nota de la práctica, se debe seguir una de estas dos opciones:
  - Utilizar EXCLUSIVAMENTE caracteres ASCII (el alfabeto inglés) en vuestros ficheros, incluídos los comentarios. Es decir, no poner acentos, ni ñ, ni ç, etcétera.
  - Entrar en el menú de Eclipse Edit $->$ Set Encoding, aparecerá una ventana en la que hay que seleccionar UTF-8 como codificación para los caracteres.

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.
- El tiempo estimado por el corrector para ejecutar cada prueba debe ser suficiente para que finalice la ejecución correctamente, en otro caso se invoca un proceso que obliga a la finalización de la ejecución de esa prueba y se considera que la misma no funciona correctamente.
- 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.
- La nota del apartado de documentación supone el 10 % de la nota de la práctica.

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 `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
    - `p01.java`: programa 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.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
  ```