Curso 2017-2018
Estructuras de Datos y Algoritmia
Práctica 3 (28 de noviembre)

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:

- Implementación de una adaptación de árbol 4-ario de objetos de tipo `PLoc`.
- Uso del tipo `TreeMap`, `TreeSet` y `Set` del API de Java.
- Aplicación de los tipos de datos anteriores, y los necesarios de las prácticas anteriores, para resolver problemas de búsqueda.

Fechas importantes

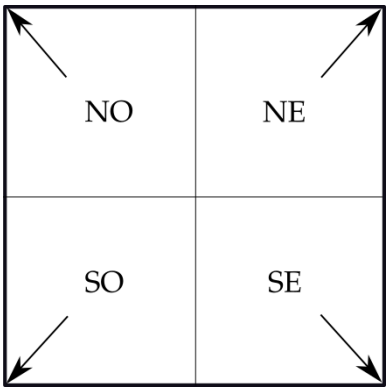
- Plazo de entrega de la práctica: desde el lunes 18 de diciembre hasta el **viernes 22 de diciembre de 2017** (más información al final de este documento).

1. Tipo Arbol

Al igual que hicimos con las listas, en esta práctica vamos a construir diferentes árboles para almacenar objetos de tipo `PLoc` con el mismo comportamiento pero diferentes estructuras de datos. Por ello definiremos el tipo `Arbol` que contendrá los métodos comunes a todos los árboles, y que puede ser un interfaz o una clase abstracta, de la que heredarán o implementarán las demás clases:

- `public void leeArbol(String f)`: se leerá el árbol 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 de fichero y se irá insertando toda la información en las variables correspondientes definidas en cada clase, según el criterio de inserción definido en cada estructura. 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 esVacio()`: nos indica si el árbol está vacío.

- `public boolean inserta(PLoc p)`: inserta un objeto `p` según los requerimientos definidos en cada una de las estructuras que lo implementen y que se definirán después [1].
- `public boolean ciudadEnArbol(String v)` devuelve `true` si encuentra alguna localidad cuya ciudad coincida con `v`, y `false` en caso contrario.
- `public TreeSet<String> getCiudades(PLoc p)`: devuelve el `TreeSet` con las ciudades asociadas al país contenido en la `PLoc` pasada por parámetro.
- `public PLoc busquedaLejana(String s)`: se busca la localidad que está más alejada en la dirección que se pasa por parámetro (`NO, NE, SO, SE`) [2]. Si es
  - `NO`, se busca la que está a la mayor latitud y menor longitud
  - `NE`, se busca la que está a la mayor latitud y mayor longitud
  - `SO`, se busca la que está a la menor latitud y menor longitud
  - `SE`, se busca la que está a la menor latitud y mayor longitud



2. Clase NodoAG

La clase `NodoAG` con la que construiremos el nodo del árbol, que será una clase `privada` de la clase `ArbolG` contendrá

- las variables de instancia siguientes:
  - `private PLoc pd;`
  - `private NodoAG no;`
  - `private NodoAG so;`
  - `private NodoAG ne;`
  - `private NodoAG se;`

[1] no habrá etiquetas repetidas

[2] teniendo en cuenta que nos centraremos en las coordenadas decimales

- y los siguientes métodos de instancia:
  - `public NodoAG()`: inicializa las variables de instancia a valores por defecto (`null`).
  - `public NodoAG(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 ArbolG

La clase **ArbolG** que tiene que implementar o heredar del tipo `Arbol`, contendrá

- las variables de instancia siguientes:
  - `private NodoAG pr;`
- y los siguientes métodos de instancia:
  - `public ArbolG()`: inicializa las variables de instancia a sus valores por defecto (`null`).
  - `public boolean inserta(PLoc p)`: crea un nodo con un objeto de tipo `PLoc` y lo añade al árbol según el siguiente criterio [3]:
    - tenemos 4 posibles subárboles a los que ir:
      - ◇ si la longitud de `p` es menor que la del nodo en que nos encontramos, entonces decido entre los subárboles `no` y `so` dependiendo del valor de la latitud (`so` si es menor y `no` si es mayor o igual);
      - ◇ si la longitud de `p` es mayor o igual que la del nodo en que nos encontramos, entonces decido entre los subárboles `se` y `ne` dependiendo del valor de la latitud (`se` si es menor y `ne` si es mayor o igual);
    - una vez elegido el subárbol correspondiente pueden pasar dos cosas:
      1. que el subárbol elegido esté vacío, entonces añadimos el nodo;
      2. que el subárbol elegido no esté vacío, entonces hay que repetir el proceso en el siguiente nivel (de forma iterativa o recursiva);

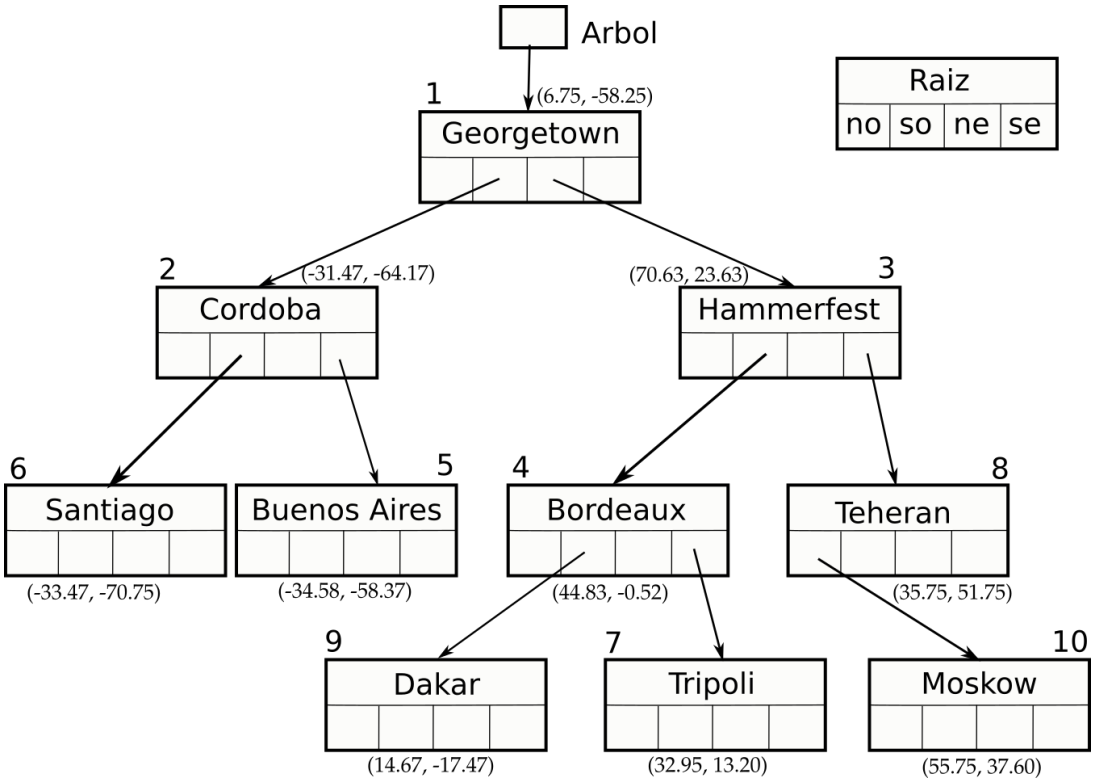
    Si el fichero contiene la siguiente información:

    SA#Guyana#Georgetown#6 45 N#58 15 O
    SA#Argentina#Cordoba#31 28 S#64 10 O
    EU#Norway#Hammerfest#70 38 N#23 38 E
    EU#France#Bordeaux#44 50 N#0 31 O
    SA#Argentina#Buenos Aires#34 35 S#58 22 O

[3] teniendo en cuenta que nos centraremos en las coordenadas decimales

SA#Chile#Santiago#33 28 S#70 45 O
AF#Libya#Tripoli#32 57 N#13 12 E
AS#Iran#Teheran#35 45 N#51 45 E
AF#Senegal#Dakar#14 40 N#17 28 O
AS#Russia#Moscow#55 45 N#37 36 E

El árbol que se obtiene es el siguiente:



- `public void recorridoInorden()`: escribe las ciudades almacenadas en el árbol siguiendo el algoritmo de recorrido en inorden con la siguiente adaptación a este árbol especial que sólo tiene una etiqueta en el nodo y 4 hijos: `no, so, Raiz, ne, se` [4]

  En el ejemplo del dibujo sería:

  Santiago
  Cordoba
  Buenos Aires
  Georgetown
  Dakar
  Bordeaux
  Tripoli

[4] en los `PLoc` no habrán ciudades con valor `null` para ninguno de los recorridos que se realicen

Hammerfest
Moscow
Teheran

- `public void recorridoNiveles()`: escribe el árbol siguiendo el algoritmo de recorrido por niveles con el siguiente formato:

  En el ejemplo del dibujo sería:

  Georgetown
  Cordoba
  Hammerfest
  Santiago
  Buenos Aires
  Bordeaux
  Teheran
  Dakar
  Tripoli
  Moskow

3. Clase ArbolS

La clase `ArbolS`, que tiene que implementar o heredar del tipo `Arbol`, contendrá:

- las variables de instancia siguientes [5]:
  - `private TreeMap<String, TreeSet<PLoc>> tm;` //Pais y localidades de ese país, respectivamente
- y los siguientes métodos de instancia [6]:
  - `public ArbolS()`: crea un árbol vacío (sin ningún elemento).
  - `public boolean inserta(PLoc p)`: añade como clave el país contenido en la `PLoc` pasada como parámetro, y como valor actualiza el `TreeSet` asociado a él con la ciudad [7].
  - `public boolean borraPais(String p)`: quita del árbol el país `p` pasado como parámetro, devolviendo `true` si se ha podido realizar el borrado y `false` en caso contrario.

[5] IMPORTANTE: `PLoc` tiene que implementar la interfaz `Comparable<PLoc>`

[6] hay que buscar en el API de java los métodos de la clase `TreeMap` y `TreeSet` más adecuado para implementar estos métodos

[7] recuerda que un mismo país puede tener varias ciudades en la base de datos y que la clave es única

- `public Set<String> getPaises()`: devuelve el conjunto de países contenidos en el `TreeMap`.

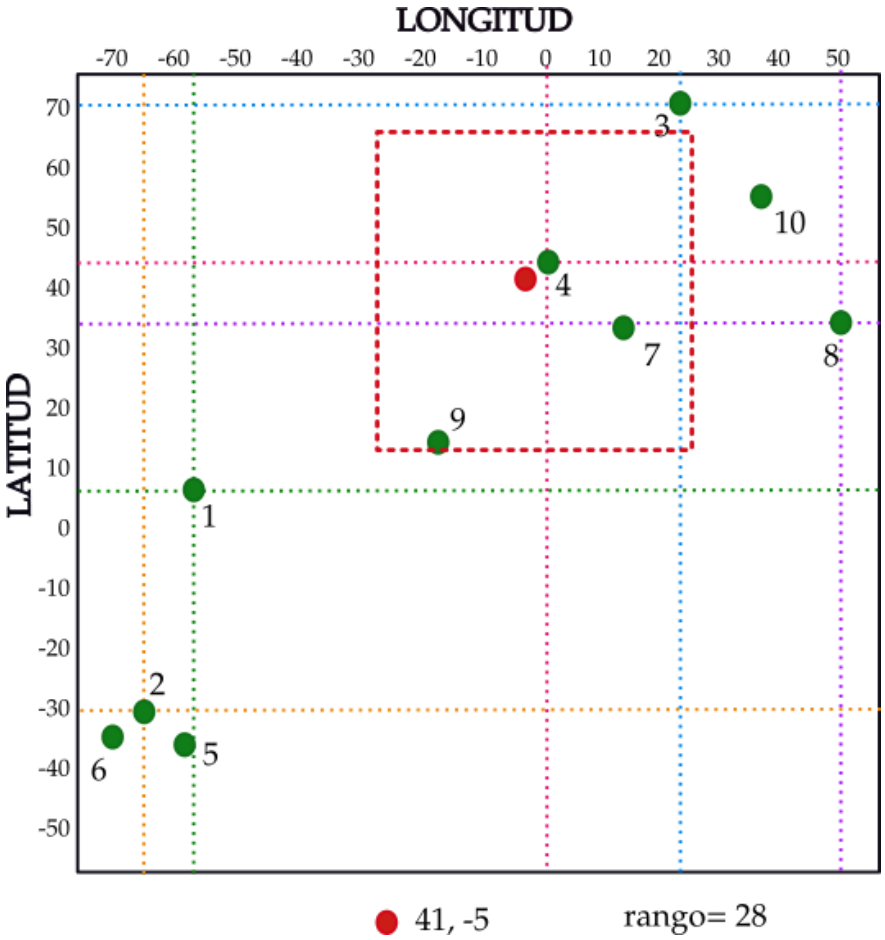
4. Búsqueda de ciudades y regiones

Vamos a extender la primera aplicación realizada en la práctica 2 a una más realista en la que ya utilizaremos la latitud y longitud para realizar búsqueda de ciudades y regiones.

A partir de una base de datos de localidades de todo el mundo en el formato de coordenadas sexagesimales, se trata de almacenar la base de datos leída en los dos tipos de datos implementados: el `ArbolG` y el `ArbolS`.

Implementa una clase denominada `BuscaLocalizacion2` donde se ejecutará la aplicación. En la clase se implementará un método main que:

1. detectará los parámetros de la aplicación, que podrá tener 4 o 2:
   - **Búsqueda por latitud y longitud**, es cuando tiene 4 parámetros:
     - nombre del fichero de la base de datos
     - latitud y longitud (formato double)
     - rango (formato double)
   - **Búsqueda por país**, es cuando tiene 2 parámetros:
     - nombre del fichero de la base de datos
     - nombre de un país
2. abre un fichero de texto con la base de datos de ciudades en formato sexagesimal, y lo almacenas en el tipo de árbol correspondiente dependiendo del número de parámetros.
3. **Búsqueda por latitud y longitud.** En este caso se han leído 4 parámetros. Esta búsqueda debe ser realizada sobre el `ArbolG`.
   - La aplicación debe mostrar todos los datos de las ciudades que se encuentran en la ventana cuadrada centrada en el valor de la longitud y latitud, y de lado igual a dos veces el valor del rango. Sobre el árbol de la página 2, la búsqueda consiste en avanzar por el árbol según el cuadrante en el que se encuentre el elemento consulta respecto a la ciudad almacenada en el nodo:



Si el paso de parámetros es con los valores `file.dat 41 -5 28`, hay que mostrar los datos de todas las ciudades cuyo valor de la latitud esté entre 13 y 69 y su longitud esté entre -33 y 23.

En concreto, cuando hagamos

```
BuscaLocalizacion2 ejemplo.db 41 -5 28
```

la salida debería mostrar `escribeInfoGps()` de las localidades (una por línea) que se encuentran dentro del rango (incluído el propio rango), ordenadas según el orden establecido para la clase `PLoc`, es decir, primero por longitud decimal, segundo por orden alfabético de la ciudad.

En el ejemplo visto sería:

```
AF - Senegal - Dakar - 14.67 - -17.47
EU - France - Bordeaux - 44.83 - -0.52
AF - Libya - Tripoli - 32.95 - 13.20
```

Si por ejemplo el rango hubiese sido 2, no habría ninguna salida y se enviaría el mensaje `NO HAY SALIDA` a la salida estándar.

ATENCIÓN: se valorará que la búsqueda en el árbol no sea exhaustiva [8].

4. **Búsqueda de países**. En este caso se han leído 2 parámetros y la base de datos está almacenada en un `ArbolS`, cuya clave de búsqueda es el país. Entonces:

   - si no está en el `ArbolS`, hay que mostrar el árbol completo con el siguiente formato [9]:

     ```
     Chile (1): Santiago
     Spain (3): Alicante - Barcelona - Madrid
     USA (2): Los Angeles - New York
     ```

   - si el país está en la base de datos, sólo hay que mostrar el país correspondiente con sus ciudades, según el formato anterior.

6. 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 `ArbolG`:
    - `public void leeArbol(String s)`
    - `public boolean inserta(PLoc p)`
    - `public TreeSet<String>getCiudades(PLoc p)`
    - `public PLoc busquedaLejana(String s)`
  - de la clase `ArbolS`:
    - `public void leeArbol(String s)`
    - `public boolean inserta(PLoc p)`
    - `public PLoc busquedaLejana(String s)`
    - `public boolean ciudadEnArbol(String s)`

[8] si la búsqueda es exhaustiva la puntuación de las pruebas correspondientes será la mitad

[9] Los países ordenados alfabéticamente, seguido del número de ciudades del país en el árbol, entre paréntesis, dos puntos, y las ciudades del país ordenadas alfabéticamente y separadas por " - "

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 del árbol (número de nodos), y consideramos que en general es `n`.
Ejemplo:

```
COSTES ArbolG:
1) int esVacio(): O(1)
2) boolean inserta(PLoc p):
3) int ciudadEnArbol(String v):
4) PLoc busquedaLejana(String s)

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

NOTA. 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 18 de diciembre hasta el **viernes 22 de diciembre**.
- Se debe entregar la práctica en un fichero comprimido los siguientes ficheros:
  - `PLoc.java`
  - `Coordenada.java`
  - `CoordenadaExcepcion.java`
  - `Arbol.java`
  - `ArbolG.java`
  - `ArbolS.java`
  - `BuscaLocalizacion2.java`

  y ningún directorio de la siguiente manera

  `tar cvfz practica3.tgz PLoc.java Coordenada.java Arbol.java ArbolG.java, ArbolS.java, CoordenadaExcepcion.java BuscaLocalizacion2.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` ⇒ NO
  `DNI 23433224 MUNOZ PICO, ANDRES` ⇒ 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 `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 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
  ```