PRÁCTICA 3

Utilización de mapas en Java

Objetivos

- Estudiar los conjuntos y los mapas para la resolución de diferentes problemas. Tanto desde el punto de vista del almacenamiento de datos como de las consultas sobre ellos.
- Aprender a utilizar la composición de estructuras de datos asociativas en la resolución de problemas.

Requerimientos

Para superar esta práctica se debe realizar lo siguiente:

- Dominar las estructuras de datos asociativas `TreeSet`, `TreeMap`, `HashSet` y `HashMap`, y la combinación de ellas para la resolución de problemas.
- Conocer cómo realizar consultas sobre estructuras de datos asociativas y la mejor forma de plantearlas en problemas sencillos.
- Resolver un problema típico de *concordancia*, haciendo uso de `TreeMap` y `TreeSet`.

Enunciado

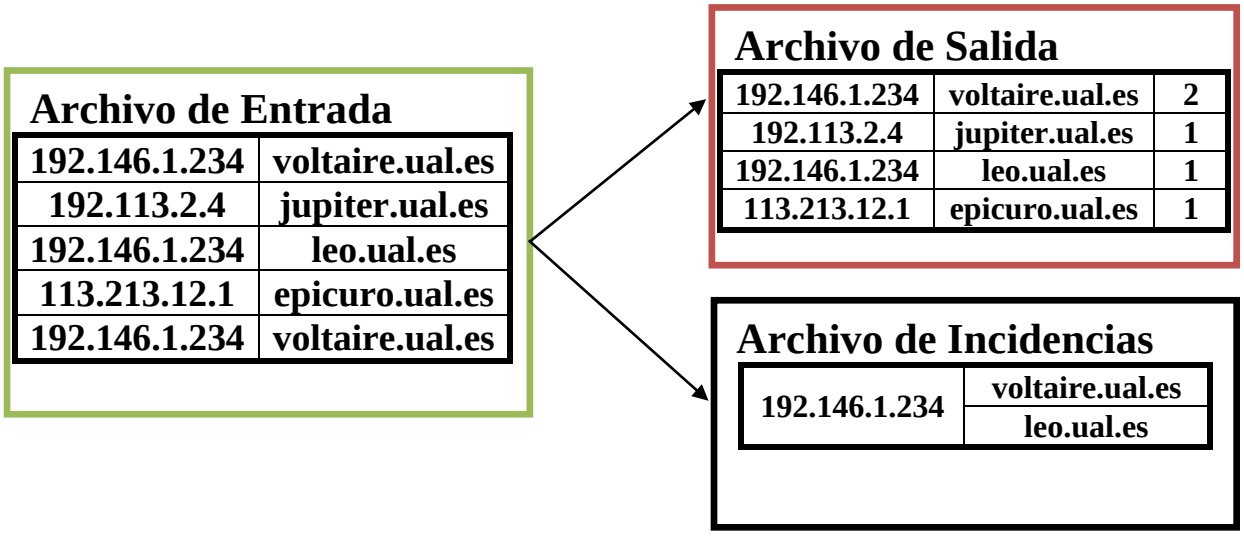
1. **Ejercicio con mapas**. Control de una puerta de acceso a una red.

Tal y como se ha estudiado en la práctica 02, las direcciones IP identifican de forma unívoca todos y cada uno de los dispositivos (ordenadores, impresoras, etc.) conectados en Internet. Cada IP tiene asociado un nombre de máquina; por ejemplo, *filabres.ual.es* tiene asociada la dirección IP *150.214.156.2*. Estas direcciones están formadas por cuatro campos que representan partes específicas de Internet, *red.subred.subred.máquina*, que el ordenador trata como una única dirección IP. Esta dirección consta de 32 bits, aunque normalmente se representa en notación decimal, separando los bits en cuatro grupos de 8 bits cada uno, expresando cada campo como un entero decimal (0..255) y separando los campos por un punto. En el ejemplo anterior: *filabres.ual.es* $\Leftrightarrow$ *150.214.156.2*.

Supongamos que se establecen conexiones desde una red a través de una puerta de acceso a otra red, y cada vez que se establece una conexión, la dirección IP del ordenador del usuario se almacena en un archivo junto con el nombre de la máquina. Estas direcciones IP y sus máquinas se pueden recuperar periódicamente para controlar quiénes han utilizado la puerta de acceso y cuántas veces lo han hecho.

Como **EJERCICIO** se pide lo siguiente: implementar un programa en Java que lea desde un archivo estas direcciones IP y los nombres de las máquinas asociadas, almacenando en un **TreeMap** de **TreeMap** de la JCF de Java. Cada una de las entradas del primer **TreeMap** guarda la dirección IP (clave) y en el otro **TreeMap** (valor), almacenando el nombre de máquina (clave) y el número de veces que dicho par (dirección IP, nombre máquina) aparece en el archivo de entrada (valor).

- Según se lee cada dirección IP y nombre de máquina del archivo, el programa debe comprobar si dicha dirección está ya en el mapa. Si lo está, debe comprobar que la máquina está como clave en el segundo mapa, en cuyo caso su contador se incrementa en 1; en caso contrario, se inserta en el mapa. Si no está la dirección, hay que crear el segundo mapa y añadir el par máquina y contador a 1.
- Una vez que todas las direcciones IP y nombres de máquina han sido leídas del archivo, se generará como resultado un archivo de salida donde se indiquen los pares (dirección IP, nombre máquina) y sus contadores.
- Finalmente, el programa deberá comprobar, para cada dirección IP, si ésta ha tenido asociado más de un nombre de máquina, en cuyo caso se generará también un archivo de salida indicando dicha incidencia.

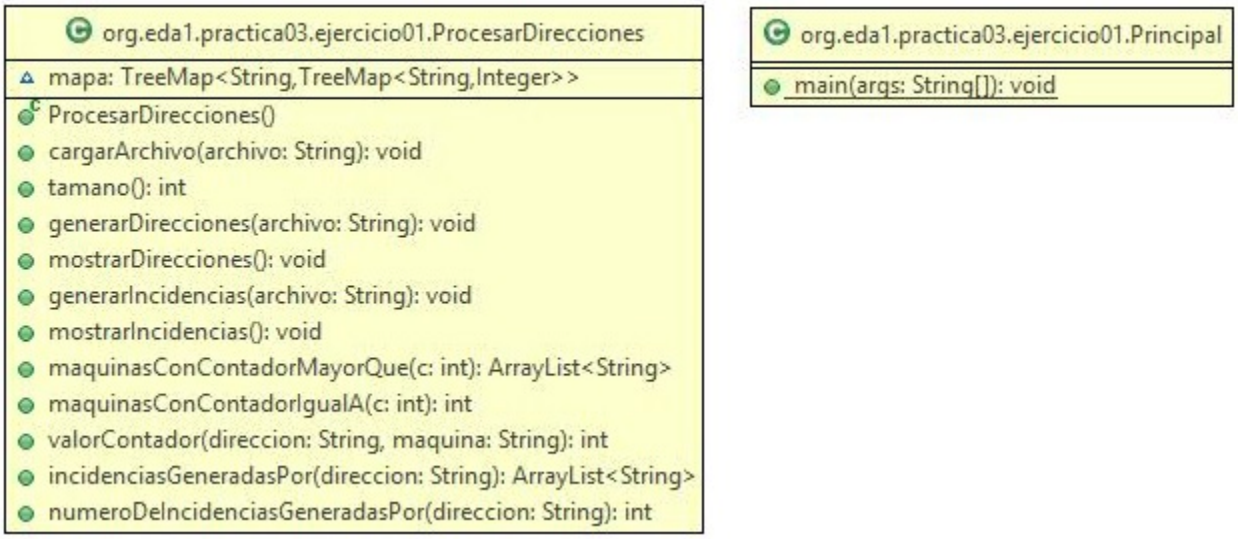


Como archivo de entrada (entrada.txt) habrá varios según se establezca en el test.

Adicionalmente, interesante indicar cómo tendría que plantearse la resolución del problema utilizando como estructura de datos en su implementación un **TreeMap<String, TreeSet<MaquinaContador>>**, en lugar de un TreeMap de TreeMap como se ha tenido que resolver **TreeMap<String, TreeMap<String, Integer>>**. Todo ello en un archivo PDF **practica03_ejercicio01** en la carpeta **Memorias**.

Además, se debe responder (con la implementación de la correspondiente función) a las siguientes consultas sencillas (asociadas a los test propuestos):

- Devolver todas las máquinas con contador mayor que 1 y mayor que 2.
- ¿Cuántas máquinas tienen un valor de contador igual a 2, e igual a 3?.
- ¿Cuál es el valor del contador del par (dirección, máquina) = (192.146.1.233, pascal.ual.es) y (dirección, máquina) = (192.146.1.234, voltaire.ual.es)?.
- Devolver las incidencias que ha registrado la dirección 192.146.1.233 y las que ha registrado la dirección 192.146.1.234 a nivel de máquinas?.
- ¿Cuántas incidencias ha registrado las direcciones 113.213.12.1 y 192.146.1.234?.



2. Ejercicio con colecciones asociativas un poco más complejas. Ciudades donde empresas de software desarrollan sus proyectos.

Como ya hemos estudiado en las prácticas 01 y 02, disponemos de información para gestionarla en una estructura de datos. Ésta está relacionada con las empresas de software que desarrollan proyectos en determinados lugares donde tienen las sedes de dichos proyectos. Para este ejercicio tendremos un el archivo de entrada "**masNuevasEmpresasProyectosCiudades.txt**".

Como **EJERCICIO** se pide lo siguiente: implementar un programa en Java que lea desde un archivo de entrada la anterior lista de **Empresa_Software Proyecto_Software Ciudad_Donde_Se_Desarrolla**, almacenando en una estructura de datos basada en **AVLTree**.

- Según se lee cada línea del archivo (**Empresa_Software Proyecto_Software Ciudad_Donde_Se_Desarrolla**), el programa debe comprobar si dicha tripleta está ya en el contenedor. Si lo está, no hacer nada pues será una línea repetida; en caso contrario, se inserta en la estructura de datos.
- Una vez que todas las líneas (tripletas) han sido leídas del archivo de entrada y almacenadas en memoria en estructura de datos asociativas **TreeMap** y **TreeSet**, se generará como resultado un listado que debe coincidir con el que se proporciona en el test.

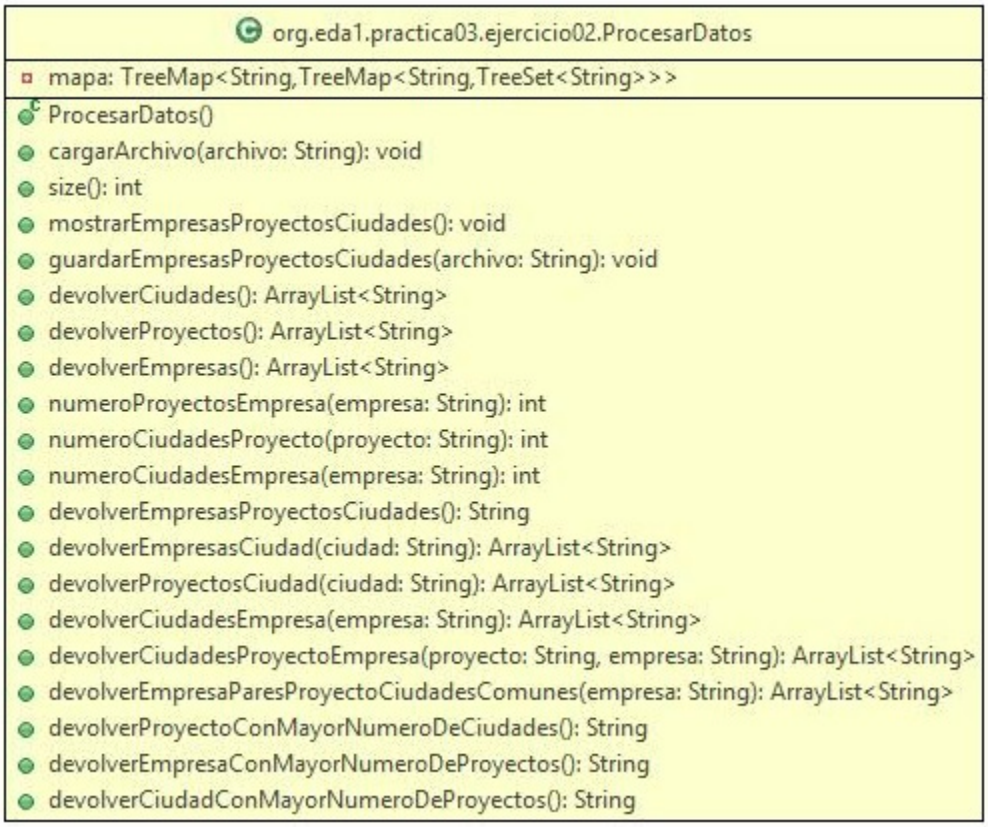
Como sugerencia para la realización del ejercicio, podemos plantear una estructura de datos basada en **TreeMap** y **TreeSet**. De forma genérica podría plantearse lo siguiente (TreeMap de TreeMaps de TreeSets)

TreeMap(Empresas, **TreeMap**(Proyectos, **TreeSet**(Ciudades)))

Además, una vez en organizados los datos en la ED indicada anteriormente, se debe responder (con la implementación de la correspondiente función) a las siguientes consultas:

- Devolver todas las empresas, todos los proyectos y todas las ciudades en tres funciones independientes.
- Devolver las empresas que tienen su sede en la ciudad Miami.
- Devolver los *proyectos* con sede en Washington.
- ¿En cuántas ciudades diferentes se desarrollan proyectos de **Google**?.
- Devolver las ciudades, donde desarrollándose el proyecto *Word*, se desarrollan a su vez proyectos de otras empresas distintas de *Micrsoft*.
- Devolver los proyectos de la empresa *Oracle*, que tienen alguna sede (ciudad) común, indicando también el nombre de dicha ciudad. Haced lo mismo para las empresas *Microsoft* y *Google*.
- Devolver cuál es el proyecto con mayor número de sedes (ciudades).
- Devolver cuál es la empresa con mayor número de proyectos.
- Devolver cuál es la ciudad con mayor número de proyectos

Para comprobar la correcta realización de este ejercicio se proporcionará el **test** que se deberá pasar correctamente.


org.eda1.practica03.ejercicio02.ProcesarDatos
mapa: TreeMap<String,TreeMap<String,TreeSet<String>>>
ProcesarDatos()
cargarArchivo(archivo: String): void
size(): int
mostrarEmpresasProyectosCiudades(): void
guardarEmpresasProyectosCiudades(archivo: String): void
devolverCiudades(): ArrayList<String>
devolverProyectos(): ArrayList<String>
devolverEmpresas(): ArrayList<String>
numeroProyectosEmpresa(empresa: String): int
numeroCiudadesProyecto(proyecto: String): int
numeroCiudadesEmpresa(empresa: String): int
devolverEmpresasProyectosCiudades(): String
devolverEmpresasCiudad(ciudad: String): ArrayList<String>
devolverProyectosCiudad(ciudad: String): ArrayList<String>
devolverCiudadesEmpresa(empresa: String): ArrayList<String>
devolverCiudadesProyectoEmpresa(proyecto: String, empresa: String): ArrayList<String>
devolverEmpresaParesProyectoCiudadesComunes(empresa: String): ArrayList<String>
devolverProyectoConMayorNumeroDeCiudades(): String
devolverEmpresaConMayorNumeroDeProyectos(): String
devolverCiudadConMayorNumeroDeProyectos(): String

3. Construir una concordancia utilizando `TreeMap` y `TreeSet`.

Una **concordancia** es una herramienta software que lee un archivo de texto y extrae todas las palabras junto con las líneas en las que dichas palabras aparecen. Los compiladores a menudo proporcionan una concordancia para evaluar el uso de identificadores (incluyendo palabras clave) en un archivo de código fuente. El programa que se pide en este ejercicio va a implementar dicha concordancia y es un buen ejemplo del uso de mapas.

Una concordancia acepta un **String** como parámetro que especifica el nombre del archivo de código fuente. Dicho programa extraerá todos los identificadores en una lista ordenada alfabéticamente, que incluye el identificador, el número de líneas que contienen el identificador en dicho archivo fuente y los números de línea en los que aparece el identificador. Un **identificador** es un **String** que empieza con una letra (”A”-“Z”, “a”-“z”) y puede continuar con cero o más caracteres que pueden ser letras o dígitos numéricos (“0”-“9”).

La primera concordancia que se pide debe mostrar la información para un identificador de acuerdo con el siguiente formato,

identificador n: l_1 l_2 l_3 … l_n

donde **n** es el número de líneas que contienen el identificador y $\mathbf{l_i}$ ($i > 0$) es la lista de números de línea en la que éste aparece.

La segunda concordancia que se pide debe modificarse el algoritmo implementado para la anterior concordancia, generando una nueva función (`newConcordance`), para registrar cada *ocurrencia de un identificador*. Se puede utilizar el mismo archivo fuente que para la anterior concordancia. Para este caso la concordancia generará como salida la información para un identificador de acuerdo con el siguiente nuevo formato:

identificador n: $l_1(m_1)$ $l_2(m_2)$ $l_3(m_3)$ $l_4(m_4)$ …

donde **n** es el número de ocurrencias del **identificador** en el archivo de código fuente y $\mathbf{l_i(m_i)}$ ($i \geq 1$) es la lista de números de líneas distintas en la que éste aparece ($\mathbf{l_i}$) con $\mathbf{m_i}$ indicando el número de ocurrencias del identificador en la línea $\mathbf{l_i}$.

A continuación vamos a intentar aclarar algunos aspectos del diseño del programa que se pide en este ejercicio para la primera de las concordancias propuestas. Una concordancia podría utilizar como estructura de datos un **TreeMap** de entradas de la forma **<String, TreeSet<Integer>>**. Un identificador de tipo **String** sería el componente clave de una entrada y el conjunto de números de línea (**TreeSet<Integer>**) sería el componente valor. Mediante la utilización de un **TreeMap**, podemos gestionar y mostrar los identificadores (clave) en orden ascendente. Además, por medio de la utilización de un **TreeSet** para el componente valor de una entrada podemos garantizar que múltiples ocurrencias de un identificador en una línea se registran una sola vez y los números de línea se gestionarán y mostrarán en orden ascendente.

Para la segunda concordancia se podría utilizar como estructura de datos también un **TreeMap**, pero ahora las entradas deben ser de la forma **<String, TreeMap<Integer, Integer>>**.

El algoritmo de concordancia utiliza un objeto **Scanner** como entrada línea a línea del archivo de código fuente. Además, en este ejercicio introduciremos conceptos de la clase **Pattern** para extraer identificadores de la línea de entrada. Como ya hemos mencionado anteriormente, cada identificador es el componente clave de una entrada en el mapa. Para mantener el número de línea actual con el que se va procesando en archivo de código fuente, podemos utilizar una variable de tipo entero, todo ello para controlar en todo momento el número de línea actual que se va procesando. Después de extraer un identificador, el algoritmo debe actualizar el mapa. Este proceso implica el acceso al mapa para determinar si el identificador está ya incluido como una entrada. En el caso de que no lo esté, el algoritmo inserta una nueva entrada con el identificador como componente clave. En cualquier caso, tanto si no está el identificador, el número de línea actual debe añadirse como un objeto **Integer** en el componente valor (**TreeSet**) de la entrada.

De forma general el algoritmo para gestionar la concordancia involucra la implementación de tres tareas principales: (1) Reconocer un identificador, (2) actualizar el mapa y (3) mostrar el resultado de la concordancia en un formato determinado; las cuales pasamos a describir brevemente a continuación.

- **Reconocer un identificador**. Para ello, vamos a repasar algunos conceptos relacionados con la clase **Pattern** para extraer identificadores de una línea de entrada. La API de Java ofrece explicaciones bastante detalladas de cómo extraer cadenas que establecen un match con una expresión regular. El proceso de forma muy resumida es el siguiente. Una expresión regular que define un identificador es **"[a-zA-Z][a-zA-Z0-9]*"**. Esta expresión nos dice que un identificador empieza con una letra en minúscula o mayúscula ([a-zA-Z]) seguido por cero o más ocurrencias de una letra (en minúscula o mayúscula) o un dígito numérico ([a-zA-Z0-9]*). El método estáticos de la clase **Pattern compile()** convierte una expresión regular a un objetos **Pattern**. Por ejemplo,
  ```
  Pattern identifierPattern = Pattern.compile("[a-zA-Z][a-zA-Z0-9]*");
  ```
  Además el método **matcher()** combinado con el objeto **Pattern**, toma una secuencia de caracteres (**String**) como argumento y crea un objeto **Matcher** que puede escanear la secuencia de caracteres e identificar subsecuencias que "casan" (que establecen un emparejamiento (match)) con el patrón.
  ```
  Matcher matcher = identifierPattern.matcher(inputLine);
  ```
  Con todo ello, en el objeto **matcher** se almacena una lista de identificadores. Éste utiliza el método **find()** para localizar el siguiente identificador, que es, la siguiente subsecuencia que "casa" (establece el emparejamiento) con la expresión regular. Dicho método devuelve false si no hay ningún match; y true si el identificador puede ser extraído como una subcadena con un rango de índice denotado por **matcher.start()** y **matcher.end()**.
  ```
  while(matcher.find()) { identifier = inputLine.substring(matcher.start(), matcher.end(); }
  ```
  Como ejemplo, su la línea de entrada (inputLine) es "value = 2 * arr[i];" entonces

  Primer find(): start() = 0 end() = 5 $\Rightarrow$ "value" = inputLine.substring(0, 5)
  Segundo find(): start() = 12 end() = 15 $\Rightarrow$ "arr" = inputLine.substring(12, 15)
  Tercer find(): start() = 16 end() = 17 $\Rightarrow$ "i" = inputLine.substring(16, 17)

- **Actualizar el mapa**. Sabemos que utilizamos el identificador como clave para el método **get()** con el objetivo de acceder al valor de una entrada en el mapa (**TreeMap**). Si el valor devuelto por dicho método es **null**, el identificador no está en el mapa. Por tanto, se debe de crear una nueva entrada en el **TreeMap** con clave el identificador y como valor un **TreeSet** vacio. En cualquier caso, hay que añadir el número de línea al conjunto y utilizar el método **put()** para actualizar el mapa (**TreeMap**). El proceso de actualización del mapa ocurre cada vez que se extrae un nuevo identificador de la línea de procesamiento del archivo de código fuente.
- **Mostrar el resultado de la concordancia en un formato determinado**. Para esta tarea podemos crear un nuevo método que se denomine **writeConcordance()** que muestre las concordancias que vamos buscando. Este método hace uso de la vista de la colección como un conjunto de entradas y un iterador para recorrerlas. El método **getKey()** de **Map.Entry** proporciona acceso al identificador. Por otro lado, el método **getValue()** de **Map.Entry** devuelve el conjunto de números de línea en las que aparece el identificador. El tamaño del conjunto es el número de números de línea, y un listado de dicho conjunto proporciona los números de línea en orden ascendente. Un esqueleto de dicho procedimiento se indica a continuación:

```
Set<Map.Entry<String,TreeSet<Integer>>> entries = map.entrySet();
TreeSet<Integer> lineNumberSet;
Iterator<Map.Entry<String,TreeSet<Integer>>> iter = entries.iterator();
Iterator<Integer> setIter;
     while (iter.hasNext()) {
          Map.Entry<String,TreeSet<Integer>> e = iter.next();
          // …
          lineNumberSet = e.getValue();
          // …
          setIter = lineNumberSet.iterator();
          while (setIter.hasNext()){
               //…
          }
          // …
     }
```

Para comprobar la correcta realización de este ejercicio se proporcionará el **test** que se deberá pasar correctamente.