

EI1013/MT1013 ESTRUCTURA DE DATOS
BOLETÍN DE PROBLEMAS II
USO DE LAS CLASES DE LA *Java Collection Framework*

En este boletín se recopilan ejercicios de examen. Todos los ejercicios han aparecido en algún examen final de la asignatura. Éste que se indica en la línea final del enunciado. Además, en cada ejercicio se indica el peso que tuvo ese ejercicio sobre la nota máxima del examen, siempre sobre 10 puntos. Para una parte de los ejercicios se incluyen posibles soluciones. Los ejercicios están de ordenados de más reciente a más antiguo.

2.1 (3 puntos) Deseamos crear el código para gestionar la matriculación de los estudiantes en las asignaturas y grupos de un curso universitario. Para ello vamos implementar la clase **MonitorMatricula**. Esta contendrá información sobre las asignaturas y los grupos disponible de éstas. Cada asignatura constará de la siguiente información.

- **Código.** Un **String** con el código único de la asignatura. P. ej. "**EI/MT1013**" o "**IR2116**", etc.
- Una colección con los **grupos** disponibles para cada asignatura.

Cada grupo de una asignatura vendrá definido por la siguiente información:

- **Código.** Un **String** con el nombre del grupo. P.ej. "**TE1**", "**LA2**", "**PR3**", etc. Una misma asignatura no contendrá dos grupos con el mismo código.
- **Horario** del grupo. Un **String** que representa el día de la semana y las horas en las que se impartirán las clases para ese grupo. P.ej. "**Lunes 12:30–14:30**".
- **Límite** de matrícula. El numero máximo de estudiantes que se permite matricular en ese grupo. Será un entero mayor que cero.
- **Matriculas.** Una colección con el nombre de todos los estudiantes matriculados en ese grupo. Cada nombre se almacenará mediante **String**.

- (a) **(0,5 puntos)** Define la parte privada de la clase **MonitorMatricula** que permita almacenar la información descrita anteriormente de la forma más eficiente. Añade un constructor por defecto que incialize las estructuras seleccionadas.
- (b) **(1 punto)** Escribe un método de la clase que permita crear un nuevo grupo y añadirlo a los datos de una asignatura dada

```
boolean crearGrupo( String asignatura, String grupo,
                     int limite, String horario )
```

Si la asignatura no existe se creará una nueva. En el caso de que el grupo ya exista el método no hará nada y devolverá **false**. Devolverá **true** en caso contrario.

- (c) **(1,5 puntos)** Escribe un método para matricular a un estudiante en un listado de grupos.

```
boolean matricular( String nombre, List<String> asignaturas,
                     List<String> grupos )
```

nombre contendrá el nombre del estudiante que se desea matricular, **asignaturas** y **grupos** son dos listas igual tamaño tales que en el posición *i*-ésima contendrán la asignatura y el grupo en la que el estudiante desea matricularse. El método deberá comprobar si hay plazas disponibles en ese grupo (no se ha llegado al límite). Si es así se matriculará

al alumno en ese grupo y se borrará la asignatura y el grupo de las listas **asignaturas** y **grupos**. De esta forma, al terminar la ejecución del método, las listas contendrán las asignaturas y grupos en las que el estudiante no ha podido ser matriculado por estar llenos.

Al tiempo que se realizan las matriculas se comprobará si existe solapamiento entre los horarios de los grupos. El método devolverá **true** si existe algún solapamiento entre alguno de los horarios en los que se ha podido matriculado al estudiante, **false** si no es así. En caso de solapamiento la matricula se realizará igualmente.

Puedes asumir que todos los datos proporcionados en los parámetros son correctos, es decir, que las listas tienen la misma longitud, que no hay repeticiones, que todas las asignaturas y grupos existen, etc.

Para poder comprobar los solapamientos podrás hacer uso del método **seSolapan** que comprueba si dos horarios se solapan y devuelve **true** si es así, y **false** si no.

```
boolean seSolapan(String h1, String h2)
```

Curso 2020/21 Examen Final 2^a Convocatoria

Solución:

```
public class MonitorMatricula {
    Map<String, Integer> limites;
    Map<String, String> horarios;
    Map<String, List<String>> matriculas;

    public MonitorMatricula() {
        limites = new HashMap<>();
        horarios = new HashMap<>();
        matriculas = new HashMap<>();
    }

    public boolean crearGrupo(String asignatura, String grupo, int limite, String h
        String idGrupo = asignatura + grupo;

        if (limites.containsKey(idGrupo))
            return false;

        limites.put(idGrupo, limite);
        horarios.put(idGrupo, horario);
        matriculas.put(idGrupo, new LinkedList<String>());

        return true;
    }

    public boolean matricular(String nombre, List<String> asignaturas, List<String>
        boolean solapamientos = false;
        Set<String> aceptados = new HashSet<>();

        Iterator<String> itAsignatura = asignaturas.iterator();
        Iterator<String> itGrupo = grupo.iterator();

        while (itAsignatura.hasNext()) {
            String id = itAsignatura.next() + itGrupo.next();

            if (limites.get(id) > matriculas.get(id).size()) {
```

```

        matriculas.get(id).add(nombre);
        itAsignatura.remove();
        itGrupo.remove();
        String h1 = horarios.get(grupo);
        for (String h2 : aceptados)
            if (seSolapan(h1, h2))
                solapamientos = true;
        aceptados.add(h1);
    }
}
return solapamientos;
}

```

Una solución alternativa sin "idea feliz"

```

public class MonitorMatricula2 {
    class Asignatura {
        String codigo;
        Map<String, Grupo> grupos;
    }

    class Grupo {
        String codigo;
        int limite;
        String horario;
        List<String> matriculas;
    }

    Map<String, Asignatura> mAsignaturas;

    public MonitorMatricula2() {
        mAsignaturas = new HashMap<>();
    }

    public boolean añadirGrupo(String asignatura, String grupo, int limite, String ho
        Asignatura asig = mAsignaturas.get(asignatura);
        if (asig == null) {
            asig = new Asignatura();
            asig.codigo = asignatura;
            asig.grupos = new HashMap<>();
            mAsignaturas.put(asignatura, asig);
        }

        Grupo gr = asig.grupos.get(grupo);
        if (gr == null) {
            gr = new Grupo();
            gr.codigo = grupo;
            gr.limite = limite;
            gr.horario = horario;
            gr.matriculas = new LinkedList<>();
            asig.grupos.put(grupo, gr);
            return true;
        }

        return false;
    }
}

```

```
public boolean matricular(String nombre, List<String> asignaturas, List<String> g
    boolean solapamientos = false;
    Set<String> aceptados = new HashSet<>();

    Iterator<String> itAsignatura = asignaturas.iterator();
    Iterator<String> itGrupo = grupo.iterator();

    while (itAsignatura.hasNext()) {
        Asignatura asig = mAsignaturas.get(itAsignatura.next());
        Grupo gr = asig.grupos.get(itGrupo.next());

        if (gr.limite > gr.matriculas.size()) {
            gr.matriculas.add(nombre);
            itAsignatura.remove();
            itGrupo.remove();
            for (String h2 : aceptados)
                if (seSolapan(gr.horario, h2))
                    solapamientos = true;
            aceptados.add(gr.horario);
        }
    }

    return solapamientos;
}
```

2.2 (3 puntos) Deseamos escribir un generador automático de exámenes. Dispondremos de un banco de cuestiones tipo test y al generar un examen se elegirá una parte de ellas de forma aleatoria. Cada cuestión constará de un código **String** único, un enunciado **String** y dos o más de posibles respuestas **String**. Cada cuestión tendrá una única respuesta correcta. La primera respuesta del listado de posibles respuestas será la correcta.

Una cuestión se representa mediante la siguiente clase:

```
public class Cuestion {
    public String codigo;
    public String enunciado;
    public List<String> respuestas;
}
```

Deseamos implementar los siguientes métodos:

- (a) (2 puntos) Escribe el método **generar_examen** que creará un examen a partir del banco de cuestiones, **banco**. Las cuestiones se elegirán de forma aleatoria y las respuestas se reordenarán también de forma aleatoria.

```
... generar_examen(int cantidad, ... banco)
```

Debes elegir el tipo de estructura que devolverá teniendo en cuenta que debe contener las cuestiones elegidas y las respuestas de las cuestiones. Será lo que verá el estudiante examinado. También debes elegir el tipo del parámetro **banco**, que contendrá todas las cuestiones posibles.

El parámetro **cantidad** indica el número de cuestiones que debe tener el examen generado. Si no hay suficientes cuestiones en el banco el método devolverá **null**. No debe haber cuestiones repetidas en el examen y las respuestas deben estar barajadas dentro de cada cuestión.

- (b) (1 punto) Escribe un método **corregir** para calcular la nota obtenida en un examen individual

```
double corregir_examen ( ... respuestas, ... banco, double correcta,
                        double incorrecta, double blanco)
```

Debes elegir la estructura de **respuestas** teniendo en cuenta que debe contener las cuestiones y las repuestas dadas por un estudiante a cada una de las cuestiones. También debes elegir el tipo del parámetro **banco**, que deberá coincidir con el apartado anterior, y que almacena la colección completa de cuestiones.

Los parámetros **correcta**, **incorrecta**, y **blanco** son las puntuaciones que se le darán a cada cuestión dependiendo de que la respuesta sea correcta, incorrecta o se ha haya dejado blanco. El resultado será la suma total de todas las respuestas.

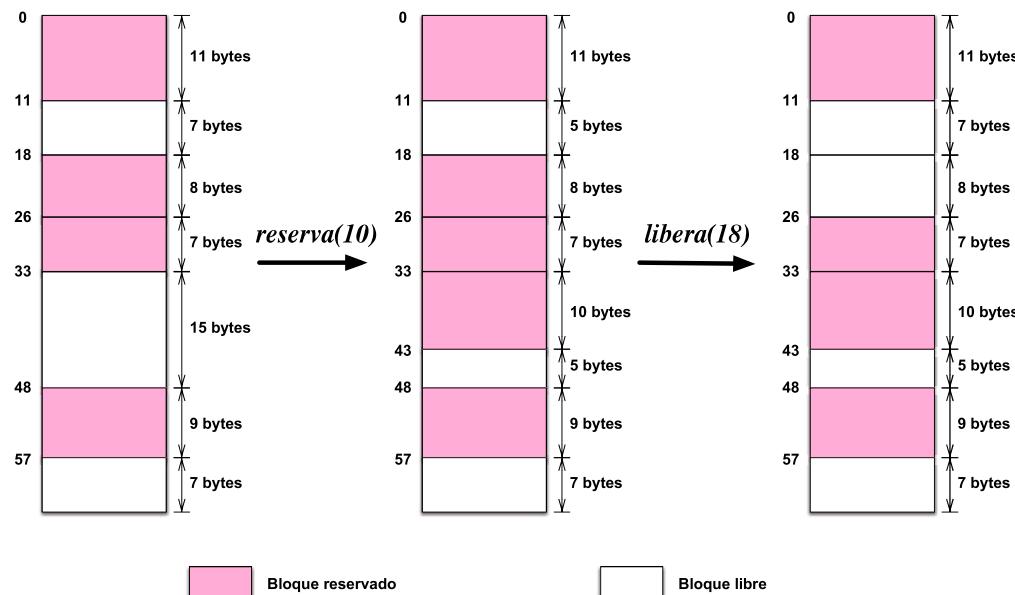
Para resolver los apartados anteriores puedes hacer uso de los siguientes métodos:

- **int aletorio (int maximo):** Devuelve un número aleatorio entre 0 y **maximo** – 1.
- **void barajar (List<T> l):** Reordena aleatoriamente los elementos de la lista **l**.

2.3 (3 puntos) En la ejecución de un programa, el gestor de memoria es el sub-proceso que se encarga de gestionar y repartir la memoria disponible entre los distintos sub-procesos que la necesiten. Deseamos implementar un gestor de memoria cuyo funcionamiento básico responda a las siguientes reglas.

- La memoria disponible se divide en bloques de tamaño variable. Cada uno de ellos comienza en una dirección de memoria *base* y tienen un *tamaño* en bytes determinado.
- Los bloques pueden estar *libres* o *reservados*, dependiendo de si han sido cedidos a un sub-proceso o no.
- Inicialmente sólo existirá un único bloque libre que empezará en la dirección 0 y ocupará toda la memoria disponible.
- Cuando un sub-proceso necesita una cantidad de memoria para su trabajo realiza una petición por esa cantidad al gestor de memoria. El gestor toma el mayor bloque libre disponible. Si el tamaño del bloque es justo la cantidad solicitada marca el bloque como *reservado*. Si es mayor de lo necesario, el gestor divide el bloque en dos, uno con la cantidad solicitada y el otro con la restante. El primero es marcado como *reservado* y el segundo sigue *libre*. A esta operación se la denomina *reservar* un bloque de memoria.
- Cuando un sub-proceso ya no necesita un bloque de memoria reservado para él se lo indica al gestor de memoria y éste libera ese bloque marcándolo como *libre*.

A continuación podéis ver un ejemplo gráfico en el que sobre una memoria de 64 bytes dividida entre distintos bloques, por cada bloque se indica su dirección de comienzo y el tamaño.



La operación **reserva (10)** encuentra el bloque más grande, en la posición 33, y lo divide en dos. La operación **libera (18)** marca como libre el bloque que comienza en la posición 18.

Dedidimos implementar la clase **GestorMemoria** para almacenar y gestionar esta información.

Completa la clase **GestorMemoria** con los siguientes métodos:

- (a) **(1 punto)** Define la parte privada de la clase eligiendo las estructuras más adecuadas para optimizar los métodos de los siguientes apartados.

Implementa también el constructor **GestorMemoria(int cantidad)**. La memoria disponible inicial vendrá indicada por el parámetro **cantidad**.

- (b) **(1 punto)** **int reserva (int cantidad) throws ExpcionMemoriaLlena**

Reservará un bloque con la cantidad de memoria indicada. Devolverá la dirección de memoria donde empiece el bloque reservado, o la excepción **ExpcionMemoriaLlena** si no ha sido posible reservarla.

- (c) **(1 punto)** **void libera (int direccion) throws ExpcionDireccionInvalida**

Liberará el bloque que comience en la dirección indicada. Si no existe un bloque que comience en esa dirección lanzará una excepción de tipo **ExpcionDireccionInvalida**.

Curso 2019/20 Examen Final 1^a Convocatoria

2.4 (2,5 puntos) Queremos construir el índice de una biblioteca. La biblioteca está compuesta por libros de los que se guarda la siguiente información:

- **Código ISBN.** Sera único. Sin embargo, la biblioteca podrá tener una o más copias de un mismo libro las cuales tendrán el mismo ISBN. En ese caso se almacenará también el número de copias.
- **Título.**
- **Autores.** Un libro podrá tener uno o más autores.
- Todos los datos anteriores son de tipo texto.

Deseamos que el índice esté optimizado para añadir, buscar y eliminar los libros según su ISBN, y para obtener un listado ordenado de los libros de cada autor.

(a) (1 punto) Define la parte privada del clase **Biblioteca** para almacenar toda esta información. Escribe también el constructor por defecto.

(b) (1 punto) Implementa un método publico de la clase que tome los datos de un libro (ISBN, título, autores) y lo añada a la biblioteca.

```
public void addBook(String Title, List<String> authors, String ISBN)
```

(c) (0.5 puntos) Implementa un método que tome el nombre de un autor y escriba por pantalla un listado con los títulos de los libros de ese autor ordenados alfabéticamente.

```
public void addBook(String name)
```

Curso 2018/19 Examen Final 2^a Convocatoria

2.5 (2 puntos) Deseamos implementar una clase **Despensa** que represente la despensa de un restaurante. La clase almacenará una colección de ingredientes, identificados mediante el nombre, y la cantidad de cada uno de ellos.

- (a) **(0.5 puntos)** Implementa la parte privada de la clase **Despensa** con las estructuras de datos más apropiadas para resolver el ejercicio de forma eficiente. Implementa también un constructor por defecto de la clase, y un método público

```
public void reponer(Collection<String> cesta)
```

Este método añadirá a la despensa todos los ingredientes que aparezcan en la colección **cesta**; por cada ingrediente que aparezca incrementará en 1 la cantidad en la despensa. Un mismo al puede aparecer más de una vez, y puedes asumir que no hay **nulls** en la colección.

- (b) **(1 punto)** Añade un método público a la clase **Despensa** que compruebe si contiene una colección de ingredientes.

```
public Collection<String> buscar(Collection<String> pedido)
```

Pedido es una colección de ingredientes identificados por su nombre. Pueden aparecer ingredientes repetidos y cada aparición significa que se buscara una unidad de ese ingrediente. Por ejemplo el pedido `["tomate", "huevo", "lechuga", "tomate", "huevo", "huevo"]` indica que se han de buscar dos tomates, tres huevos y una lechuga. El método devolverá una colección con los ingredientes que faltan. En el ejemplo anterior, faltan solo dos huevos, el resultado será `["huevo", "huevo"]`. En caso de que haya suficiente cantidad de todos los ingredientes del pedido devolverá una colección vacía. Puedes asumir que no hay **nulls** en **pedido**. En este método el contenido de la despensa no se puede modificar.

- (c) **(0.5 puntos)** Añade un método público a la clase **Despensa** un método público que sirva un pedido de ingredientes.

```
public boolean servir(Collection<String> pedido)
```

Tomará una colección **pedido** con las mismas características que la del apartado anterior y actualizará los datos internos de **Despensa** para restar los ingredientes servidos. Devolverá **true** en caso de que en la despensa hubiera suficiente cantidad de los ingredientes servidos; y **false** en el caso de no hubiera suficiente cantidad de algún ingrediente. En este último caso el pedido no se sirve y no se modifican las cantidades de los ingredientes en la despensa.

Curso 2018/19 Examen Final 1^a Convocatoria

2.6 (2 puntos) Un banco posee una serie de cuentas en la que los clientes almacenan una cantidad de dinero. Una de las operaciones más habituales es la transferencia de dinero entre cuentas del banco. En una transferencia existe una cuenta de origen que transfiere una cantidad a una cuenta destino. Al realizar la operación, el saldo de la cuenta de origen se reduce en la cantidad transferida, mientras que el saldo de la cuenta de destino se incrementa en la misma cantidad.

Deseamos escribir una clase **Banco** que guarde el saldo de las cuentas existentes y un histórico de las transferencias producidas entre ellas. Una cuenta tiene un código, **String**, y un saldo, **double**. No puede haber cuentas con códigos repetidos. Para representar una transacción entre cuentas definimos la clase **Transferencia**:

```
public class Transferencia implements Comparable<Transferencia>{
    public String origen;
    public String destino;
    public String fecha; // Formato: "AAAAAMMDD"
    public double cantidad;

    public Transferencia(String origen, String destino,
                         String fecha, double cantidad) { ... }

}
```

(a) **(1 punto)** Escribe la parte privada de la clase **Banco** eligiendo las estructuras más eficientes posibles para el resto de los métodos del ejercicio. Añade, además el constructor:

```
public Banco(List<String> codigos, List<Double> saldos)
```

Tomará dos listas, una con códigos de cuenta y otra con los saldos respectivos, y los guardará. En caso de que las listas sean de distinto tamaño devolverá una excepción **IllegalArgumentException**. Si aparecen códigos repetidos, se almacenará sólo uno de ellos con la suma de todos los saldos. Puede haber cuentas con saldos negativos.

(b) **(1 punto)** Añade a la clase el método **contabiliza** que procesa una transferencia. Esto es, modifica los saldos de la cuenta origen y destino según la cantidad transferida y almacena la transferencia para futuras consultas.

```
public boolean contabiliza(Transferencia tr)
```

El método lanza una excepción **IllegalArgumentException** en el caso de que alguna de las cuentas no exista. El método no realizará ni almacenará la transferencia en el caso de que la cuenta *origen* no disponga de suficiente saldo positivo para realizar la transferencia. Tampoco lo hará si la cantidad a transferir es negativa. Devolverá **false** en estos dos últimos casos y **true** si se realiza la transferencia con éxito.

Curso 2017/18 Examen Final 2^a Convocatoria

2.7 (3 puntos) Deseamos escribir una aplicación para la gestión del almacén de un supermercado. En este almacén se guardarán distintas cantidades de una gran variedad de productos. Cada producto se identificará mediante un código único formado por letras y números. Se creará la clase **Supermercado** para guardar el inventario de los productos almacenados en el almacén y la cantidad y precio de cada uno de ellos.

- (0.5 puntos) Define la clase **Producto** para que permita almacenar el código, la cantidad y el precio de un producto. Incluye los constructores y métodos que creas necesarios.
- (0.5 puntos) Escribe la parte privada de la clase **Supermercado** y un constructor por defecto de la clase. Elige las estructuras más eficientes posibles.
- (1 punto) Cuando llega al almacén un camión cargado de productos para el supermercado es necesario actualizar el inventario del almacén. Añade un método **almacenar** que tome un listado con los productos que contiene el camión y actualice las cantidades en el inventario (los precios no se actualizan).

```
public void almacenar(List<Producto> camion)
```

En la lista pueden aparecer productos repetidos o productos nuevos. En caso de que la lista sea **null** se debe lanzar una excepción **IllegalArgumentException**.

- (1 punto) Cuando un cliente realiza una compra toma una cantidad variable de diversos productos. Añade un método **calcularRecibo** que calcule el valor de su compra y actualice el inventario del almacén.

```
public double calcularRecibo(List<Producto> compra)
```

Tomará una lista con los productos comprados y las cantidades de estos. Se calculará el precio total. En el caso de que no hubiera suficiente cantidad en el almacén de un determinado producto, se modificará **compra** para limitarla a la cantidad disponible de ese producto. Por otro lado se actualizará el almacén para reflejar las cantidades que se han sacado en la compra. En el caso de que la lista sea **null** se deberá lanzar una excepción **IllegalArgumentException**.

Curso 2017/18 Examen Final 1^a Convocatoria

2.8 (3 puntos) Deseamos escribir una aplicación para gestionar la ocupación de camas y pacientes en un hospital. En estos las camas se distribuyen en habitaciones. Dependiendo del tamaño de las habitaciones, en cada una de ellas hay un número distinto de camas. Las camas pueden estar vacías u ocupadas por pacientes. Las camas no se numeran, pero las habitaciones sí, con un código de habitación. Nuestra aplicación debe mantener el control de las camas libres y ocupadas, dónde se hayan los pacientes y de asignar o liberar camas a medida que lleguen o marchen pacientes.

Más concretamente, guardaremos información de todas las habitaciones del hospital. Por cada habitación guardaremos, al menos, su código y el nombre de los pacientes que ocupan las camas.

- (1 punto) Define las estructuras internas de la clase **GestionHospital** que almacenen toda la información referente a la gestión de camas. Elige las clases adecuadas para simplificar y optimizar los métodos que se solicitan en los siguientes apartados.
- (0,5 puntos) Escribe un constructor para **GestionHospital** que tome como parámetro un diccionario que contenga los códigos de todas las habitaciones y el número de camas en cada una de ellas, **Map<String, Integer>**. Puedes asumir que los datos contenidos están bien formados, es decir que no hay códigos **null** ni habitaciones con 0 o menos camas.

```
public void GestionHospital(Map<String, Integer> listado)
```

- (c) **(1 punto)** Añade dos métodos a la clase. Uno tal que dado el código de una habitación devuelva un listado con los nombres de los pacientes en esa habitación. La lista estará vacía si no hay pacientes. Y devolverá **null** si no existe una habitación con el código indicado.

```
public List<String> PacientesHabitacion(String código)
```

El otro método tomará el nombre de un paciente y devolverá el código de la habitación en la que se encuentre. Devolverá **null** si el paciente no se haya ingresado en el hospital.

```
public String HabitacionPaciente(String nombre)
```

- (d) **(0,5 puntos)** Añade un método que ingrese un nuevo paciente. Tomará su nombre, buscará una habitación con una cama libre, y asignará el paciente a esa cama. Devolverá el código de la habitación en la que se encuentre la cama o **null** en el caso de que no queden camas libres o ya exista un paciente ingresado con el mismo nombre.

```
public String IngresarPaciente(String nombre)
```

- (e) Añade un método a la clase que devuelve el porcentaje de ocupación de camas en el hospital, es decir, la relación entre el número de camas ocupadas y camas totales.

```
float Ocupacion()
```

Curso 2016/17 Examen Final 2^a Convocatoria

2.9 (3.5 puntos) Deseamos crear una clase **Competicion** que almacene los marcadores de las partidas de una competición en forma de liga. En cada competición participarán un número de equipos que permanecerá constante durante toda ella. Los equipos se irán enfrentando en partidas de dos en dos, produciendo cada una de ellas un marcador final. Deseamos que la clase almacene todas las partidas producidas en la competición hasta la fecha, optimizando la inserción y búsqueda de los marcadores de las partidas.

- (a) (0,5 puntos) Define la clase **Competicion** y su parte privada especificando las estructuras que usarás para almacenar los datos. Además debes escribir un constructor que tome como entrada el listado de equipos e inicialice las estructuras.

```
public Competicion(List<String> equipos)
```

- (b) (1 punto) Escribe un método para añadir una nueva partida y su nuevo marcador. Tomará los dos equipos de la partida y los puntos obtenidos por cada uno de ellos. Si la partida ya existía previamente no se hará nada. Devolverá un booleano indicando si la partida se ha añadido o no por ya estar incluida. El orden de los equipos es relevante para el almacenamiento de las partidas: la partida “equipo1 contra equipo2” es distinta de “equipo2 contra equipo1”.

```
public boolean partida (String equipo1, int puntos1,
                      String equipo2, int puntos2)
```

- (c) (1 punto) Escribe un método que consulte el marcador de una partida. Tomará como parámetros el nombre de los dos equipos y devolverá un objeto de la clase **Marcador** como resultado, o **null** en el caso de que la partida entre los dos equipos no se haya producido aún.

```
public class Marcador {
    public int puntos1;
    public int puntos2;
}
public Marcador marcador(String equipo1, String equipo2)
```

- (d) (1 punto) Escribe un método que calcule los puntos totales obtenidos por un equipo hasta la fecha. Para asignar los puntos de clasificación a los equipos, en cada partida se asignarán 2 puntos al equipo con más puntos en el marcador y 0 al de menos, o 1 punto a ambos si tienen los mismos puntos en el marcador.

```
public int puntuacion(String equipo)
```

Nota: Todos los métodos producirán una excepción **IllegalArgumentException** si el nombre de los equipos no existe, el de ambos es idéntico, o alguno de ellos es **null**.

Curso 2016/17 Examen Final 1^a Convocatoria

2.10 (2 puntos) Representamos un documento de texto como una lista de líneas de texto, **List <String>**.

Escribe un método público estático **index** que tome un documento representado así y devuelva una colección ordenada alfabéticamente que contenga por cada palabra distinta del documento, las líneas en las que aparezca esa palabra. Los números de líneas deberán estar ordenados de menor a mayor y no habrá repeticiones.

Nota: Puedes usar un método **List <String> split (String line)** que toma una línea y devuelve un listado con todas las palabras que aparecen en la línea en minúsculas y habiendo eliminado los signos de puntuación.

Curso 2015/16 Examen Final 1^a Convocatoria

Solución:

```
public static TreeMap<String, List<Integer>> index(List<String> doc){  
    TreeMap<String, List<Integer>> map = new TreeMap<String, List<Integer>>();  
  
    ListIterator<String> iter = doc.listIterator();  
    while (iter.hasNext()) {  
        int line= iter.nextInt();  
        for (String word:split(iter.next())) {  
            List<Integer> lines = map.get(word);  
            if (lines == null) {  
                lines = new ArrayList<Integer>();  
                lines.add(line);  
                map.put(word, lines);  
            } else  
                if (lines.get(lines.size()-1) != line)  
                    lines.add(line);  
        }  
    }  
    return map;  
}
```

2.11 (3.5 puntos) Queremos implementar las clases necesarias para representar un árbol de directorios de un sistema de ficheros. Un árbol de directorios es una estructura jerárquica en la que un directorio contiene una colección de contenidos. Cada uno de estos contenidos puede ser un archivo de datos u otro directorio (subdirectorio). El siguiente ejemplo muestra un caso de un árbol de directorios.

```

root\
    README.txt
    configure.sh
    data\
        base.dat
        align.cfg
    src\
        cpp\
            aligner.cpp
            main.cpp
        py\
            setup.py

```

El directorio root contiene dos archivos, README.txt y configure.sh, y dos subdirectorios data y src. A su vez los subdirectorios data y src contienen diversos archivos y subdirectorios.

El nombre del archivo aligner.cpp es "aligner.cpp" y su ruta o path es "root\src\aligner.cpp". De forma similar el nombre del subdirectorio py es "py" y su ruta o path es "root\src\py".

Para cada archivo almacenamos su nombre, fecha y hora de creación. Por cada directorio almacenaremos su nombre, la colección de elementos que contiene y la fecha y hora de creación. Definimos la interfaz **Content**

```

public interface Content {
    boolean isDirectory();
    String getName();
    void setName(String name);
    String getDate();
}

```

que representa los posibles elementos contenidos en un directorio. Tendrán todos nombre y fecha de creación. El método **isDirectory** indicará si el elemento es un subdirectorio.

También definimos completamente la clase **File**, que implementa la interfaz **Content** y almacena la información referida a un archivo individual.

```

public class File implements Content {
    private String name;
    public String date;
    public File(String name) {
        this.name = name;
        this.date = getCurrentDate();
    }
}
public boolean isDirectory() {
    return false;
}
public String getName() {
    return null;
}
public String getDate() {
    return date;
}
public void setName(String name) {
    this.name = name;
}
public String toString() {
    return name + " " + date;
}

```

El método **getCurrentDate()** devuelve un **String** que representa la fecha y hora actual.

- (a) (1 punto) Define la clase **Directory** para almacenar un directorio. Esta clase deberá implementar la interfaz **Content**. Define los datos de la parte privada, el constructor y los métodos de la interfaz.

- (b) (2.5 puntos) Añade a la clase `Directory` el método `add`, que añadirá un nuevo contenido al árbol de directorios.

```
public boolean add(String path, boolean isDir);
```

El método recibe la ruta del contenido a añadir. El parámetro `isDir` indica si se desea añadir un directorio, `true`, o un archivo, `false`. No se podrán añadir contenidos en rutas que no existen ni elementos con nombres duplicados en el mismo directorio. El método devolverá un valor lógico indicando si la operación ha tenido éxito. **Nota:** La ruta del fichero no contiene el directorio origen como primer componente, ya que se supone que son relativas al directorio actual.

Como ejemplo sobre el directorio anterior

<code>add("src/configure.sh", false)</code>	<i>// Crea un nuevo archivo en 'src'</i>
<code>add("build", true)</code>	<i>// Crea un nuevo directorio en 'root'</i>
<code>add("tmp/data.txt", false)</code>	<i>// Error: la ruta no existe</i>
<code>add("data/base.dat", true)</code>	<i>// Error: nombre duplicado en 'data'</i>

Para implementar este método podrás hacer uso de los siguientes métodos auxiliares:

- `String getCurrentTime()` Devuelve un texto con la hora y fecha actual.
- `List<String> splitPath(String path)` Toma un ruta y la divide en los nombres que la componen:

```
splitPath("root/src/configure.sh") -> ["root", "src", "configure.sh"]
```

- `String pathName(String path)` Extrae el ultimo nombre de una ruta completa.

```
pathName("root/src/configure.sh") -> "configure.sh"
```

- `String joinPath(List<String> names)` Realiza la operación inversa a `splitPath`.

```
joinPath(["root", "src", "configure.sh"]) -> "root/src/configure.sh"
```

Curso 2014/15 Examen Final 2^a Convocatoria

2.12 (2 puntos) Queremos construir el índice de una biblioteca. La biblioteca está compuesta por libros de los que se guarda la siguiente información:

- **Código ISBN.** Sera único. Sin embargo, la biblioteca podrá tener una o mas copias de un mismo libro las cuales tendrán el mismo ISBN. En ese caso se almacenará también el número de copias.
- **Título.**
- **Autores.** Un libro podrá tener uno o mas autores.
- Todos los datos anteriores son de tipo texto.

Deseamos que el índice esté optimizado para añadir, buscar y eliminar los libros según su ISBN, y para obtener un listado de los libros de cada autor.

- (a) **(0.5 puntos)** Define la parte privada del clase **Biblioteca** para almacenar toda esta información. Escribe también el constructor.
- (b) **(1 punto)** Implementa un método publico de la clase que tome los datos de un libro (ISBN, título, autores) y lo añada a la biblioteca.

```
public void addBook(String Title, List<String> authors, String ISBN);
```

- (c) **(0.5 puntos)** Implementa un método que tome el nombre de un autor, y escriba por pantalla un listado con los títulos de los libros de ese autor ordenados alfabéticamente.

```
public void addBook(String name);
```

Curso 2014/15 Examen Final 1^a Convocatoria

2.13 (3 puntos) Nos han encargado gestionar las inscripciones de los alumnos nuevos de la Universitat Jaume I. En la preinscripción los alumnos deben indicar las titulaciones de la la UJI que desean cursar, para ello en el formulario indican sus preferencias por orden de prioridad. Por otro lado las titulaciones tienen un límite máximo de plazas. Nuestro objetivo es asignar cada candidato a una titulación por orden de nota y las preferencias de éste.

Disponemos de dos fuentes de información de entrada

- Una colección sin ningún orden con la información de los estudiantes candidatos: nombre, DNI, y la nota del examen de selectividad (número decimal con 2 dígitos), y un listado ordenado con los códigos de las titulaciones preferidas por ese alumno. No hay un número máximo de preferencias por alumno.
 - Un índice con los códigos de las titulaciones y el numero de plazas disponibles para cada una de ellas. El código de cada titulación es un texto.
- (a) **(1 punto)** Escribe al definición de la clase **Estudiante** que por cada uno de ellos incluirá sus datos y su lista de preferencias. Deberás implementar un constructor, un método para añadir códigos de titulación a la lista de preferencias y todos los métodos que creas convenientes, si hay alguno, para el siguiente apartado.
- (b) **(2 puntos)** Escribe un método que procesa las preinscripciones de los alumnos y los asigna a una titulación concreta. La función deberá tomar como entrada dos colecciones: una con los estudiantes preinscritos y otra con las titulaciones y el número de plazas disponibles. Elige las estructuras mas adecuadas para los parámetros de entrada. Este método deberá mostrar por pantalla un listado de las asignaciones que consistirá en DNI, nombre de alumno y titulación. El algoritmo deberá seguir los siguientes pasos:
1. En primer lugar, se ordenaran los alumnos por nota. Puedes suponer que no existen notas repetidas.
 2. Se irán seleccionando los alumnos por nota de mayor a menor. Por cada alumno se elegirá su primera opción. Si aún hay plazas disponibles de esa titulación se le asignara. Si no se pasará al siguiente alumno. Cuando se haya terminado con la primera pasada, se volverá a repetir el ciclo pero usando la segunda opción de cada alumno. Y así sucesivamente hasta haber asignado todos los alumnos.
 3. En el caso de que se agoten todas las preferencias de un alumno se le asignará el código especial "Lista de espera".
 4. Finalmente se mostraran las asignaciones por orden de DNI.

Curso 2013/14 Examen Final 2^a Convocatoria

2.14 (2.5 puntos) Vamos a implementar una base de datos de expertos en diversos temas, con el fin de que cuando necesitemos un experto en uno o varios temas concretos podemos averiguar rápidamente quienes son los expertos en esos temas. A cada experto lo identificaremos por su nombre, p.e.: "Francis Crick", y cada área de experiencia por una cadena, p.e.: "DNA", "molecular biology", "biophysics", "neuroscience".

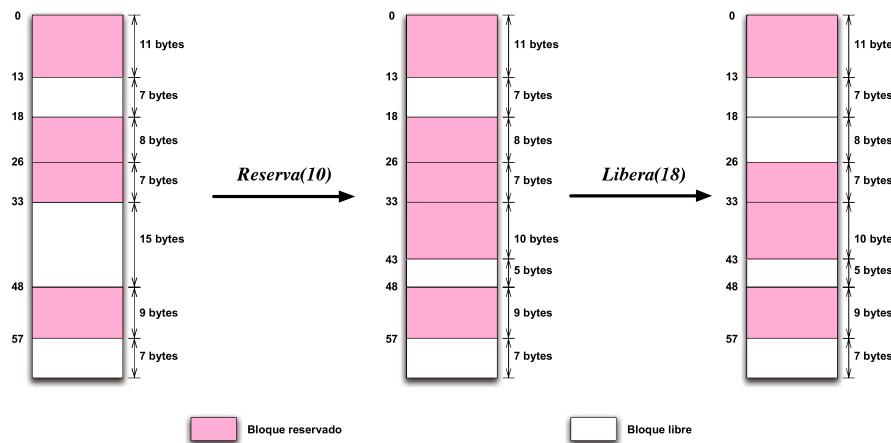
- (a) **(0.5 puntos)** Define la parte privada de la clase **ExpertsDB** e implementa el constructor. La clase deberá almacenar los datos descritos en el enunciado. Elige las estructuras más adecuadas para implementar eficientemente los métodos solicitados en los siguientes apartados.
- (b) **(1 punto)** Añade el método **addExpert** que añadirá un experto nuevo a la base de datos. Junto al nombre también se indicarán los temas de ese experto. El método deberá tener en cuenta si los temas y el experto estaban ya en la base de datos. Devolverá un valor booleano indicando si se ha añadido información nueva a la base de datos.
- (c) **(1 punto)** Añade un método **findExperts** tal que dado un tema concreto, devuelva todos los expertos que lo son de ese tema.
- (d) Añade un método **findExperts2** tal que dado un conjunto de temas devuelva todos los expertos en todos esos temas. Este apartado no apareció en el examen.

Curso 2013/14 Examen Final 1^a Convocatoria

2.15 (3 puntos) Un gestor de memoria se encarga gestionar y repartir la cantidad de memoria disponible entre los distintos programas que se lo soliciten. Deseamos implementar uno cuyo funcionamiento básico responda a las siguientes reglas.

- La memoria se divide en bloques de tamaño variable, que comienzan en una dirección de memoria *base* y tienen un *tamaño* en bytes determinado.
- Los bloques pueden estar libres u ocupados, dependiendo de si han sido reservados o no.
- Inicialmente sólo existirá un único bloque libre que empezará en la dirección cero y ocupará toda la memoria.
- Cuando se realiza una reserva de memoria, el usuario indica la cantidad de bytes que quiere reservar. El gestor examina los bloques libres en busca de uno con suficiente tamaño para la cantidad solicitada. Si encuentra uno con el tamaño justo lo marca como ocupado. Si es mayor de lo necesario, el gestor divide el bloque en dos, uno con la cantidad solicitada y el otro con la restante. El primero es marcado como ocupado y el segundo sigue libre.
- Cuando se realiza la liberación de un bloque éste es marcado como libre.

A continuación podéis ver un ejemplo gráfico en el que sobre una memoria de 64 bytes ya segmentada se realizan una operación de reserva y otras de liberación.



Implementa la clase **GestorMemoria**. Define los campos y las estructuras auxiliares necesarias (*pista: utilizar una estructura de datos para los bloques libres y otra para los bloques ocupados*).

La clase contará con los siguientes métodos públicos.

- **GestorMemoria(int cantidad)**
El constructor tomará el tamaño de la memoria y la pondrá toda como libre en un único bloque.
- **int Reserva(int cantidad) throws ExpcionMemoriaLlena**
Reservará un bloque con la cantidad de memoria indicada. Devolverá la dirección de memoria donde empieza el bloque reservado, o la excepción **ExpcionMemoriaLlena** si no ha sido posible reservarla.
- **void Libera(int direccion) throws ExpcionDireccionInvalida**
Liberará el bloque que comience en la dirección indicada. Si no existe un bloque que comience en esa dirección lanzará una excepción de tipo **ExpcionDireccionInvalida**.

2.16 (3 puntos) Deseamos crear un clase que almacene los resultados de las elecciones generales por provincias y partidos, de tal forma que por cada provincia se almacenen los datos obtenidos por cada partido presentado en esa provincia. El número de partidos presentados en cada provincia puede ser variable, y no todos los partidos se presentan en cada provincia. Las estructuras elegidas para representar los datos deberán ser las más eficientes computacionalmente para resolver todas los métodos que se piden en el ejercicio. **Lee detenidamente todos los apartados antes de empezar la solución.**

- (a) Define una clase **EleccionesGenerales** que almacene los resultados de todas las provincias. Los métodos deberán optimizar el acceso a los datos por el nombre de provincia. Deberás definir el constructor por defecto de la clase y los siguientes métodos básicos:

- **public void IncluirProvincia (String provincia).**
Añadirá una nueva provincia a la estructura de datos.
- **public void VotosPartido(String Provincia , String partido , int votos).**
Guardará los votos obtenidos por el partido en esa provincia.

- (b) Añade un método a la clase que dado el nombre de una provincia muestre por pantalla todos los partidos que han obtenido votos en dicha provincia, así como el número de votos y el porcentaje respecto al total de votos de la provincia. Los partidos deberán aparecer ordenados alfabéticamente.

```
public void MostrarVotosProvincia(String provincia);
```

- (c) Añade un método que dado un partido calcule el número de votos totales.

```
public int VotosTotal(String partido);
```

- (d) Añade un método que devuelva una colección con todos los totales de todos los partidos.

Curso 2012/13 Examen Final 1^a Convocatoria

2.17 (1,5 puntos) Dos listas son *equivalentes* si contienen los mismos elementos aunque no necesariamente en el mismo orden. Implementa un método estático que reciba dos listas y devuelva **true** si son equivalentes y **false** en caso contrario.

Curso 2011/12 Examen Final 2^a Convocatoria

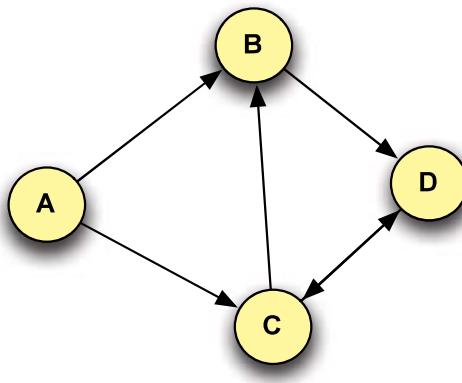
2.18 (2 puntos) Pretendes escribir un programa que gestione una lista con las solicitudes económicas de ayudas familiares. La información de cada solicitud de ayuda constará de los datos del solicitante (nombre, NIF y domicilio) y de la cantidad solicitada. Una persona (identificada únicamente por su NIF) sólo podrá solicitar una ayuda. La secuencia de solicitudes es importante ya que se concederán por orden de llegada.

1. Define e implementa la clase **Ayuda** que contendrá, al menos:
 - Los atributos privados con la información de cada solicitud de ayuda.
 - El constructor al que se le pasarán los datos del solicitante y la cantidad de ayuda solicitada.
 - Redefinir los métodos **toString** y **equals**.
2. Define la parte privada y el constructor de la clase **SolicitudesDeAyudas** para guardar todas las solicitudes.
3. La clase **SolicitudesDeAyudas** que contendrá operaciones para:
 - Dado el NIF de un solicitante, devolver la cantidad que ha solicitado.
 - Añadir una nueva solicitud (debe controlarse si efectivamente se ha añadido o si ya existía una petición anterior para ese solicitante).
 - Listar todas las ayudas.

Completa la clase con la implementación de los métodos correspondientes.

Curso 2011/12 Examen Final 1^a Convocatoria

2.19 (2 puntos) Pretendemos implementar un grafo con arcos sin peso usando un diccionario. De esta forma un grafo con nodos de la clase **T** se implementa como un diccionario en el que cada par se compone de un elemento de clase **T** y un lista de elementos de clase de **T**. Los elementos de la lista son los nodos adyacentes desde el nodo correspondiente. Como ejemplo el grafo de la figura se describiría con los siguientes pares: (A, [B, C]), (B, [D]), (C, [B, D]), (D, [C]).



Escribe una función estática que dado un grafo, y dos nodos concretos, uno inicial y otro final, determine si existe un camino que los une. *Pista: usad un conjunto de nodos que inicialmente contenga el primer elemento, e id sucesivamente añadiendo los elementos a los que se puede llegar desde cada uno de los nodos del conjunto.*

Curso 2011/12 Examen Final 1^a Convocatoria

2.20 (1 punto) En la Java Collection Framework define la interfaz de la colección **Map** que es implementada mediante tablas de dispersión, **HashMap**, y árboles binarios de búsqueda, **TreeMap**. Razona qué ventajas y desventajas tienen cada una de estas implementaciones.

Curso 2011/12 Examen Final 1^a Convocatoria