Programación II - 2012/2013 - Universitat Jaume I
Examen final - Segunda convocatoria

10 de julio de 2013

La duración máxima de este examen es de 3 horas. No puedes consultar libros ni apuntes. Para aprobar debes obtener al menos 4 puntos en el examen final y al menos 5 puntos en la nota media del curso.

Ejercicio 1 — 0,5 puntos

Considera la siguiente implementación de las clases `Punto` y `Círculo`:

```
public class Punto {
    private int x;
    private int y;

    public Punto(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Punto(Punto p) {
        x = p.x;
        y = p.y;
    }

    public void mover(int d) {
       x += d;
       y += d;
    }

    public String toString() {
       return "(" + x + ", " + y + ")";
    }
}
```

```
public class Círculo {
    private int radio;
    private Punto centro;

    public Círculo(int r, Punto c) {
        radio = r;
        centro = c;
    }

    public Círculo(int r, int x, int y) {
        radio = r;
        centro = new Punto(x, y);
    }

    public void cambiar(int d) {
        radio += d;
        centro = new Punto(centro);
        centro.mover(d);
    }

    public String toString() {
        return "R" + radio + centro;
    }
}
```

Indica qué se escribe en la salida estándar al ejecutar el siguiente programa:

```
public class Prueba {
    public static void main(String[] args) {
        Punto punto1 = new Punto(1, 2);
        Punto punto2 = punto1;
        Círculo círculo1 = new Círculo(1, 2, 3);
        Círculo círculo2 = new Círculo(2, punto2);

        punto1.mover(1);
        círculo1.cambiar(2);
        círculo2.cambiar(3);
        punto2.mover(4);

        System.out.println("punto1: " + punto1);
        System.out.println("punto2: " + punto2);
        System.out.println("círculo1: " + círculo1);
        System.out.println("círculo2: " + círculo2);
    }
}
```

Ejercicio 2 — 0,5 puntos

Indica qué se escribe en la salida estándar al ejecutar el siguiente programa, qué problema resuelve el método `enigma` y que características deben tener los dos vectores para que ese método resuelva ese problema.

```
public class Enigma {

    public static int enigma(int[] v1, int[] v2) {
        return misterio(v1, 0, v2, 0);
    }

    private static int misterio(int[] v1, int i1, int[] v2, int i2) {
        if (i1 >= v1.length || i2 >= v2.length)
            return 0;
        else if (v1[i1] == v2[i2])
            return 1 + misterio(v1, i1 + 1, v2, i2 + 1);
        else if (v1[i1] < v2[i2])
            return misterio(v1, i1 + 1, v2, i2);
        else
            return misterio(v1, i1, v2, i2 + 1);
    }

    public static void main(String[] args) {
        int[] númerosLucas = {1, 3, 4, 7, 11, 18};
        int[] dígitosPares = {0, 2, 4, 6, 8};
        int[] dígitosImpares = {1, 3, 5, 7, 9};

        System.out.println( "A: " + enigma(númerosLucas, dígitosPares) );
        System.out.println( "B: " + enigma(númerosLucas, dígitosImpares) );
        System.out.println( "C: " + enigma(númerosLucas, númerosLucas) );

    }

}
```

Ejercicio 3 — 4,5 puntos

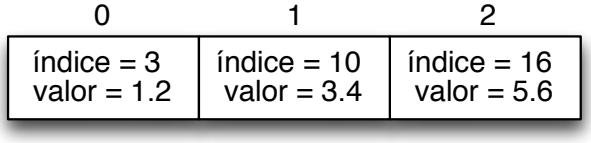
En el contexto de una aplicación de cálculo matemático necesitamos usar vectores de números flotantes con una particularidad: gran cantidad de sus elementos contienen el valor cero. Este tipo de vector recibe el nombre de *vector disperso*.

Guardar en memoria todos los elementos de un vector disperso supone desperdiciar memoria, más cuanto mayor sea la cantidad de elementos nulos. Una técnica más eficiente en consumo de memoria, aunque más ineficiente en tiempo, consiste en almacenar solamente los elementos no nulos. Para ello, podemos utilizar un vector en el que cada componente contenga el valor de un elemento no nulo del vector disperso junto con su índice (en particular, la longitud de ese vector será 0 si no hay elementos no nulos). Además, para ciertas operaciones conviene que las componentes se guarden *ordenadas de menor a mayor índice* en ese vector.

Por ejemplo, para representar el siguiente vector disperso, que tiene talla 20 y solo tres elementos no nulos (el elemento con índice 3 tiene valor 1.2, el de índice 10 vale 3.4 y el de índice 16 vale 5.6):

$$[\ 0, 0, 0, 1.2, 0, 0, 0, 0, 0, 0, 3.4, 0, 0, 0, 0, 0, 5.6, 0, 0, 0\ ]$$

podríamos utilizar un vector como el de la figura siguiente, que está formado por tres componentes, una por cada elemento no nulo del vector:



Considera que ya disponemos de la siguiente clase `Componente`:

```
public class Componente {
    private int índice;
    private double valor;
    public Componente(int elÍndice, double elValor) {
        índice = elÍndice;
        valor = elValor;
    }
    public Componente(Componente otra) {
        índice = otra.índice;
        valor = otra.valor;
    }
    public int getÍndice() {
        return índice;
    }
    public double getValor() {
        return valor;
    }
    public void setValor(double elValor) {
        valor = elValor;
    }
}
```

Haciendo uso de esa clase `Componente`, sin modificarla, queremos completar la siguiente clase `VectorDisperso`:

```
public class VectorDisperso {
    private Componente[] datos;
    private int talla;
    public VectorDisperso(int laTalla) {
        talla = laTalla;
        datos = new Componente[0];
    }
    // Aquí irían los métodos que se piden
}
```

Internamente, las componentes se deben guardar *ordenadas de menor a mayor índice* en el vector. Teniendo en cuenta eso, te pedimos implementar los siguientes métodos públicos, que se deben poder utilizar como ilustran los ejemplos. Si quieres hacer uso de cualquier otro método de la clase `VectorDisperso`, debes implementarlo también. No puedes utilizar ninguna clase de las bibliotecas del lenguaje.

En cada uno de los siguientes apartados **solo se aceptan soluciones eficientes**, que tengan en cuenta que el vector de componentes está ordenado por índice. Además, debes expresar, empleando la notación O, el tiempo de ejecución de tu solución en cada apartado.

a) [$1,5$ **puntos**] Un método `obtenerValor` que tenga como parámetro un índice i y devuelva el valor del elemento i-ésimo del vector disperso. Si el índice es menor que cero o mayor o igual que la talla del vector disperso, el método lanzará una excepción del tipo `ExcepciónFueraDeRango`[1]. En caso contrario, si existe una componente que contenga el índice i, el método devolverá el valor almacenado en esa componente; si no existe tal componente, devolverá cero.

Ejemplo: `double valor = miVectorDisperso.obtenerValor(10);`

b) [$1,5$ **puntos**] Un método `ponerACero` que tenga como parámetro un índice i y ponga a cero el elemento i-ésimo del vector disperso. Como antes, si el índice está fuera del rango correcto, el método lanzará una excepción del tipo `ExcepciónFueraDeRango`[1]. En caso contrario, si existe una componente cuyo índice es i, será eliminada (recuerda que no se almacenan los valores nulos); si no existe, no cambiará nada.

Ejemplo: `miVectorDisperso.ponerACero(10);`

c) [$1,5$ **puntos**] Un método `sumar` que tenga como parámetro otro vector disperso. Si las tallas de los dos vectores no coinciden[2], el método lanzará una excepción del tipo `ExcepciónTallasDiferentes`[1]. Si coinciden, el método devolverá un nuevo vector disperso de esa misma talla en el que el valor del elemento i-ésimo será la suma de los valores i-ésimos de los dos vectores dispersos. Los dos vectores dispersos iniciales no se deben modificar y cualquier modificación posterior en cualquiera de los tres vectores dispersos no debe afectar a ninguno de los otros dos.

Ejemplo: `VectorDisperso resultado = miVectorDisperso.sumar(otroVectorDisperso);`

[1] Considera que las clases `ExcepciónFueraDeRango` y `ExcepciónTallasDiferentes` ya están definidas.

[2] No confundas la talla de los vectores dispersos con la cantidad de elementos no nulos.

Queremos desarrollar una aplicación que nos permita gestionar un registro de facturas. Cada factura tiene un número único de factura, está adscrita a un departamento y tiene un importe determinado. Considera que ya disponemos de la siguiente clase `Factura`:

```
public class Factura {
    private int numFactura;
    private String departamento;
    private double importe;
    public Factura(int elNúmeroDeFactura, String elDepartamento, double elImporte) {
        numFactura = elNúmeroDeFactura;
        departamento = elDepartamento;
        importe = elImporte;
    }
    public int getNumFactura() { return numFactura; }
    public String getDepartamento() { return departamento; }
    public double getImporte() { return importe; }
}
```

Haciendo uso de esa clase, sin modificarla, queremos completar la siguiente clase `RegistroDeFacturas`:

```
public class RegistroDeFacturas {
    private Nodo primerNodo;
    // Aquí irían los métodos que se piden
}
```

Internamente, las facturas se guardan *ordenadas de menor a mayor por departamento*[3] en una lista doblemente enlazada de nodos en la que cada nodo tiene un dato de tipo `Factura`. Considera, además, que la definición de la clase `Nodo` incluye únicamente lo siguiente:

```
    Factura factura;
    Nodo sig, ant;
    Nodo(Factura laFactura, Nodo elNodoSiguiente, Nodo elNodoAnterior) {
        factura = laFactura;
        sig = elNodoSiguiente;
        ant = elNodoAnterior;
    }
```

Te pedimos implementar los siguientes métodos públicos, que se deben poder utilizar como ilustran los ejemplos. Si quieres hacer uso de cualquier otro método de las clases `Nodo` o `RegistroDeFacturas`, debes implementarlo también.

En cada uno de los siguientes apartados **solo se aceptan soluciones eficientes**, que tengan en cuenta que la lista de facturas está ordenada por departamento. Además, debes expresar, empleando la notación O, el tiempo de ejecución de tu solución en cada apartado.

a) [1,5 **puntos**] Un método `añadir` que tenga como parámetro un objeto de la clase `Factura` y que inserte ordenadamente (por departamento) un nuevo nodo con dicha factura en la lista de facturas.

   Ejemplo de uso:
   ```
   RegistroDeFacturas registro = new RegistroDeFacturas();
   registro.añadir(new Factura(4, "Música", 225.50));
   ```

b) [1,5 **puntos**] Un método `consultarDepartamentoMásLucrativo`, sin parámetros, que devuelva el departamento cuyas facturas sumen un importe total que sea máximo. En caso de que dos o más departamentos tengan el mismo importe total máximo, el método puede devolver cualquiera de esos departamentos. Si la lista está vacía debe devolver `null`.

   Ejemplo de uso:
   ```
   String depto = registro.consultarDepartamentoMásLucrativo();
   ```

c) [1,5 **puntos**] Un método `eliminarFacturasConDepartamento` que tenga como parámetro un departamento y elimine de la lista todas las facturas cuyo departamento coincida con ese. Por supuesto, las facturas restantes deben quedar ordenadas por departamento.

   Ejemplo de uso:
   ```
   registro.eliminarFacturasConDepartamento("Música");
   ```

Antes de entregar no olvides indicar el tiempo de ejecución en cada apartado de los ejercicios 3 y 4.

[3] Recuerda que, si `c1` y `c2` son de tipo `String`, `c1.compareTo(c2)` devuelve un número negativo, cero o un número positivo según `c1` sea menor que, igual o mayor que `c2`, respectivamente, de acuerdo con el orden lexicográfico de las cadenas.