Programación II - 2011/2012 - Universitat Jaume I
Examen final - Segunda convocatoria

4 de julio de 2012

La duración máxima de este examen es de 3,5 horas. No puedes consultar libros ni apuntes.

Ejercicio 1 — 0,8 puntos

Considera la siguiente implementación de la clase `BoletoSorteo`:

```
import java.util.Arrays;

public class BoletoSorteo {
    private String identificador;
    private int[] números;

    public BoletoSorteo(String unIdentificador, int talla) {
        identificador = unIdentificador;
        números = new int[talla];
    }
    public BoletoSorteo(BoletoSorteo otroBoleto) {
        identificador = otroBoleto.identificador;
        números = otroBoleto.números;
    }
    public void modificaNúmeros() {
        for (int k = 0; k < números.length; k++)
            números[k] = números[k] + k;
    }
    public void aumentaTalla() {
        int[] aux = new int[números.length + 1];
        for (int k = 0; k < números.length; k++)
            aux[k] = números[k];
        números = aux;
    }
    public void muestra(String etiqueta) {
        System.out.println(etiqueta + " : " + identificador + " = " + Arrays.toString(números));
    }
}
```

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

```
public class PruebaBoletoSorteo {
    public static void modificaNúmeros(BoletoSorteo[] boletos) {
        for (int i = 0; i < boletos.length; i++)
            boletos[i].modificaNúmeros();
    }
    public static void aumentaTalla(BoletoSorteo[] boletos) {
        for (int i = 0; i < boletos.length; i++)
            boletos[i].aumentaTalla();
    }
    public static void muestra(String etiqueta, BoletoSorteo[] boletos) {
        for (int i = 0; i < boletos.length; i++)
            boletos[i].muestra(etiqueta + i);
    }
    public static void main(String[] args) {
        BoletoSorteo b1 = new BoletoSorteo("K2", 3),
                     b2 = b1,
                     b3 = new BoletoSorteo(b1);
        BoletoSorteo[] misBoletos = {b1, b2, b3};

        modificaNúmeros(misBoletos);
        muestra("A", misBoletos);

        aumentaTalla(misBoletos);
        muestra("B", misBoletos);
    }
}
```

Para cada uno de los dos apartados siguientes:

- Indica qué se escribe en la salida estándar al ejecutarlo.
- Describe brevemente para qué sirve el método `enigma`.
- Expresa, empleando la notación O, el tiempo de ejecución del método `enigma` y el tiempo de ejecución total del programa.

a) **[0,6 puntos]**

```
import java.util.Arrays;

public class EnigmaUno {

    public static void main(String[] args) {
        int[] fibonacci = {1, 1, 2, 3, 5, 8, 13, 21, 34};
        enigma(fibonacci);
        System.out.println(Arrays.toString(fibonacci));
    }

    public static void enigma(int[] v) {
        misterio(v, 0, v.length - 1);
    }

    public static void misterio(int[] v, int i, int j) {
        if (i < j) {
            int aux = v[i];
            v[i] = v[j];
            v[j] = aux;
            misterio(v, i + 1, j - 1);
        }
    }

}
```

b) **[0,6 puntos]**

```
import java.util.Scanner;

public class EnigmaDos {

    public static void main(String[] args) {
        System.out.print("Dame un número entero: ");
        Scanner entrada = new Scanner(System.in);
        int n = entrada.nextInt();  // Haz la traza suponiendo que se introduce 9

        for (int i = 1; i <= n; i++)
            System.out.println("E" + i + ": " + enigma(i));
    }

    public static int enigma(int n) {
        return misterio(n, 1, 1);
    }

    public static int misterio(int n, int a, int b) {
        if (n == 1)
            return a;
        else
            return misterio(n - 1, b, a + b);
    }

}
```

Queremos desarrollar una aplicación que nos permita gestionar una agenda de tareas pendientes. Cada tarea tiene una fecha y una descripción. Considera que ya disponemos de la clase `Fecha`, que entre otros dispone de un método público `compareTo`[1], y de la siguiente clase `Tarea`:

```
public class Tarea {
    private Fecha fecha;
    private String descripción;
    public Tarea(Fecha laFecha, String laDescripción) {
        fecha = laFecha;
        descripción = laDescripción;
    }
    public Fecha getFecha() { return fecha; }
    public String getDescripción() { return descripción; }
}
```

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

```
public class Agenda {
    private Tarea[] tareas;
    public Agenda() { tareas = new Tarea[0]; }
    // Aquí irían los métodos que se piden
}
```

Internamente, las tareas se guardarán *ordenadas por fecha* en el *vector* de objetos de la clase `Tarea`. Para no consumir memoria innecesaria, la longitud del vector coincidirá en todo momento con la cantidad de tareas almacenadas (en particular, la longitud será 0 si no hay tareas). Teniendo en cuenta eso, te pedimos implementar los siguientes métodos públicos, que se deben poder utilizar como ilustran los ejemplos:

a) **[1 punto]** Un método `consultarPeorFecha`, sin parámetros, que devuelva la fecha con más tareas pendientes. Puedes tratar los empates como quieras. Si la agenda está vacía debe devolver `null`.

Ejemplo:
```
Agenda agendaTrabajo = new Agenda();
// Aquí podría ir código que inserta datos en agendaTrabajo
Fecha díaNegro = agendaTrabajo.consultarPeorFecha();
```

Solo se aceptan soluciones eficientes, que tengan en cuenta que el vector de tareas está ordenado por fecha. Expresa, empleando la notación O, el tiempo de ejecución de tu solución.

b) **[1 punto]** Un método `consultarTareas` que tenga como parámetro una fecha y devuelva como resultado un vector de cadenas que contenga las descripciones de las tareas cuya fecha coincida con esa. Como caso particular, la longitud del vector devuelto será cero si no se encuentra ninguna tarea con esa fecha.

Ejemplo:
```
String[] cosasDelDíaNegro = agendaTrabajo.consultarTareas(díaNegro);
```

Solo se aceptan soluciones cuyo tiempo de ejecución sea $O(m + \log n)$ siendo n la cantidad de tareas total y m la cantidad de tareas de la fecha pedida. Como todas las tareas con la misma fecha estarán juntas, después de encontrar cualquiera de ellas en tiempo $O(\log n)$ puedes consultarlas todas en tiempo $O(m)$.

c) **[1 punto]** Un método `eliminarTareas` que tenga como parámetro una fecha y elimine del vector todas las tareas cuya fecha coincida con esa. Por supuesto, las tareas restantes deben quedar ordenadas por fecha en un vector cuya longitud coincida con la cantidad de tareas.

Ejemplo:
```
agendaTrabajo.eliminarTareas(díaNegro);
```

Solo se aceptan soluciones cuyo tiempo de ejecución sea $O(n)$, siendo n la cantidad inicial de tareas.

d) **[1 punto]** Un método `combinarAgenda` que tenga como parámetro otra agenda y añada todas sus tareas al vector de tareas propio. Como antes, las tareas deben quedar ordenadas por fecha en un vector cuya longitud coincida con la cantidad de tareas.

Ejemplo:
```
Agenda agendaDeLuisa = new Agenda(), agendaDeJorge = new Agenda();
// Aquí podría ir código que inserta datos en las agendas
agendaDeLuisa.combinarAgenda(agendaDeJorge);
```

Solo se aceptan soluciones cuyo tiempo de ejecución sea $O(n)$, siendo n la cantidad total de tareas.

[1] Si `f1` y `f2` son de tipo `Fecha`, `f1.compareTo(f2)` devuelve un número negativo, cero o un número positivo según `f1` sea menor que, igual o mayor que `f2`, respectivamente.

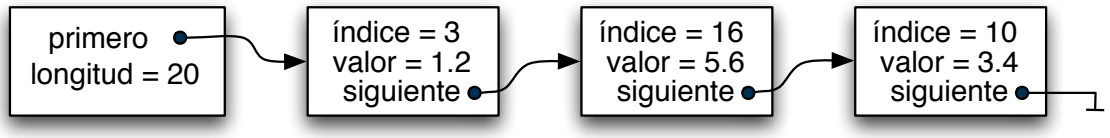
En el contexto de una aplicación de cálculo matemático necesitamos usar vectores de números flotantes con una particularidad: gran cantidad de sus elementos contienen el valor cero. Este tipo de vector recibe el nombre de *vector disperso*.

Utilizar vectores en Java para representar vectores dispersos supone desperdiciar memoria, más cuanto mayor sea la cantidad de componentes 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 una lista enlazada en la que cada nodo contenga el valor de un elemento no nulo del vector disperso junto con su índice (además de una o más referencias a otros nodos, dependiendo del tipo de lista utilizada).

Por ejemplo, para representar el siguiente vector disperso, que tiene longitud 20 y solo tres elementos no nulos (el elemento con índice 3 vale 1.2, el elemento 10 vale 3.4 y el elemento 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 una lista enlazada como la de la figura siguiente, que está formada por tres nodos, uno por cada elemento no nulo del vector:



Implementa las clases `Nodo` y `VectorDisperso` empleando esa técnica. Puedes utilizar el tipo de lista que quieras (simplemente enlazada, doblemente enlazada, con referencia al primer nodo, con referencia al primero y al último, etc.) pero no puedes utilizar vectores de Java ni ninguna clase de las proporcionadas en las bibliotecas del lenguaje. Puedes decidir también si prefieres mantener los nodos ordenados por su índice o no, siempre que tu implementación de todo lo que se te pide sea coherente con tu elección. Además de los atributos y métodos de la clase `Nodo` y de los atributos de la clase `VectorDisperso`, te pedimos implementar los siguientes métodos públicos, que se deben poder utilizar como ilustran los ejemplos. Expresa en cada apartado, empleando la notación O, el tiempo de ejecución de tu solución.

a) **[0,5 puntos]** Un constructor que tenga como parámetro la longitud del vector disperso.

   Ejemplo: `VectorDisperso vector = new VectorDisperso(20);`

   La valoración de este apartado incluye la definición de los atributos de las clases `Nodo` y `VectorDisperso`.

b) **[0,5 puntos]** Un método `obtenerValor` que tenga como parámetro un índice i y devuelva el valor del elemento i-ésimo del vector. Si el índice no es válido (mayor o igual que cero y menor que la longitud del vector) el método lanzará una excepción del tipo `ExcepcionFueraDeRango`[2]. En caso contrario, si existe un nodo que contenga el índice i, el método devolverá el valor almacenado en el nodo; si no existe tal nodo, devolverá cero.

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

c) **[1,25 puntos]** Un método `ponerACero` que tenga como parámetro un índice i y ponga a cero el elemento i-ésimo del vector. Como antes, si el índice está fuera del rango correcto el método lanzará una excepción del tipo `ExcepcionFueraDeRango`[2]. En caso contrario, si existe un nodo cuyo índice es i, será eliminado (recuerda que no se almacenan los valores nulos); si no existe, no cambiará nada.

   Ejemplo: `vector.ponerACero(10);`

d) **[1,75 puntos]** Un método `asignarValor` que tenga dos parámetros, un índice i y un valor x, y asigne el valor x al elemento i-ésimo del vector disperso. Como antes, si el índice i está fuera del rango correcto se lanzará una excepción del tipo `ExcepcionFueraDeRango`[2]. En caso contrario, si x vale cero se debe llamar al método del apartado anterior; si x no vale cero y ya existe un nodo con índice i, se actualizará su valor en ese nodo; finalmente, si x no vale cero y no existe en la lista ningún nodo cuyo índice sea i, se insertará un nuevo nodo con índice i y valor x.

   Ejemplo: `vector.asignarValor(19, 5.6);`

[2]Considera que la clase `ExcepcionFueraDeRango` ya está definida.