Examen de Estructuras de Datos y Algoritmos
(Ingeniería Informática)

Febrero 2009

Primera parte (50% nota del examen)

1) Se desea escribir un método con la cabecera que se muestra abajo, que pueda eliminar de la cola de prioridad `cola` los elementos que se indican en la lista `borrar`. El método, además de modificar `cola`, debe retornar otra cola de prioridad con los elementos que se han borrado. Ignorar los elementos de `borrar` que no estén en `cola`. Indicar la eficiencia de la operación desarrollada suponiendo que *n* es el número de elementos de `cola`, y *m* el de `borrar`, y suponiendo que la cola de prioridad está implementada por medio de un montículo binario. Intentar que esta eficiencia sea máxima, suponiendo que *n* es mucho mayor que *m*.

```
public static <E extends Comparable> PriorityQueue<E> elimina
    (PriorityQueue<E> cola, List<E> borrar)
{...}
```

2) Se dispone de la clase `MapaDoble` que tiene dos atributos que son mapas con claves del tipo `String` y valores también del tipo `String`. La clase permite almacenar relaciones uno a uno entre parejas de palabras que representan el DNI y el número de la seguridad social de una persona, con objeto de poder convertir eficientemente el uno en el otro y al revés. Para ello el primer mapa tiene claves que son DNIs y valores que son números de la seguridad social, y el segundo lo hace al revés. Escribir los métodos cuya cabecera se indica abajo, que permiten insertar una pareja de datos y borrarla, respectivamente (suponer que los constructores de las excepciones no tienen parámetros). Indicar cuál será la eficiencia de estos métodos en función del número de parejas almacenadas, *n*, suponiendo que los mapas no están muy llenos y que las claves se distribuyen homogéneamente.

```
import java.util.*;
public class MapaDoble
{
    private Map<String,String> dniAsegsocial =
       new HashMap<String,String>();
    private Map<String,String> segsocialAdni =
       new HashMap<String,String>();

    /**
     * Inserta dni y num seg social en el mapa doble
     * Lanza YaExiste si el dni o el num de la seg social ya existen
     */
    public void inserta(String dni, String segsocial) throws YaExiste
    {...}

    /**
     * Borra un dni y num seg social del mapa doble
     * Lanza NoExiste si el dni o el num de la seg social no existen,
     * o si no están ambos relacionados
     */
    public void elimina(String dni, String segsocial) throws NoExiste
    {...}

    ...
}
```

3) Se dispone de un grafo no dirigido en el que cada vértice contiene el nombre un científico, y los arcos que salen de él indican qué científicos han realizado publicaciones conjuntas con él. Por ejemplo, si hay un arco del científico *A* al *B* se entiende que *A* y *B* tienen alguna publicación conjunta, en la que ambos figuran como autores. El contenido del arco no se utiliza. El grafo cumple la interfaz `Grafo` vista en clase.

Lo que se pide es escribir el método cuya cabecera se muestra abajo que es capaz de retornar una lista de los nombres de los científicos (no repetidos) que trabajan en temas afines al científico cuyo nombre se indica en el parámetro `c`. Se entiende que trabajan en temas afines todos aquellos que tengan publicaciones conjuntas con `c` o con científicos que a su vez tengan publicaciones conjuntas con `c` (es decir, aquellos vértices que están a distancia 1 o distancia 2 de `c`).

```
public static List<String> trabajanEnTemasAfines
      (Grafo<String,Integer> publicaciones, String c)
{...}
```

Examen de Estructuras de Datos y Algoritmos
(Ingeniería Informática)

Febrero 2009

Segunda parte (50% nota del examen)

4) Se dispone de una implementación de un mapa cerrado mediante tabla *hash*, como la vista en clase. El mapa dispone de los atributos que se indican, siendo `tabla` un array que contiene objetos de la clase `Entry`. El mapa utiliza la técnica del borrado perezoso, de modo que una casilla no válida de la tabla es aquella que vale `null`, o cuyo atributo `borrado` vale `true`. Escribir la operación `containsValue()` que funcione de acuerdo con la operación del mismo nombre de la interfaz `Map`. ¿Cuál es la eficiencia de esta operación en función del número de datos del mapa, *n*, y del tamaño de la tabla, *m*?

```
import java.util.*;
public class MapaCerrado <K,V>
{
    /**
     * Clase Entry, que define una pareja clave-valor
     * y anota si se ha borrado o no
     */
    private static class Entry<K,V> {
        private K clave;
        private V valor;
        boolean borrado;

        public Entry(K clave, V valor) {
            this.clave = clave;
            this.valor = valor;
            this.borrado=false;
        }

        public boolean equals(Object otro) {
            if (otro instanceof Entry) {
                return clave.equals(((Entry<K,V>)otro).clave);
            } else {
                return false;
            }
        }
    }

    // atributos privados de la clase MapaCerrado
    private Object[] tabla;
    private int num;

    /**
     * Método containsValue
     */
    boolean containsValue(Object value)
    {...}

    ...
}
```

5) Escribir el pseudocódigo de un método que permite buscar un dato en un árbol binario **no** ordenado que sigue la interfaz del árbol binario vista en clase. El método debe retornar un iterador de árbol binario cuyo nudo actual sea el elemento encontrado, o `null` si no se encuentra en el árbol.

```
método <E> busca(ArbolBinario<E> arbol, E elem) retorna IterArbolBin<E>
```

6) Se dispone de una implementación de una lista enlazada simple como la vista en clase. Escribir el método `divide()`, cuya cabecera se indica abajo, y que sirve para partir la lista por la mitad. La primera mitad contendrá los elementos del 0 al (`num`-1)/2 (división entera, siendo `num` el tamaño de la lista) y será retornada como una nueva lista. La 2ª mitad contendrá el resto de los elementos y se quedará en el objeto actual. Hacer también un diagrama con un ejemplo de la estructura de la lista antes de llamar al método, y de las dos listas después de la llamada.

```
import java.util.*;
public class ListaEnlazadaSimple<E>
   extends AbstractSequentialList<E>
{
    // atributos privados
    private Celda<E> principio;
    private int num;

    // clase privada que define la celda
    private static class Celda<E> {
       E contenido;
       Celda<E> siguiente;

       Celda(E cont) {
          contenido=cont;
       }
    }

    /**
     * Constructor que crea la lista vacia
     */
    public ListaEnlazadaSimple()
    {
        // crea la celda cabecera
        principio=new Celda<E>(null);
        num=0;
    }

    /**
     * Divide la lista en dos
     */
    public ListaEnlazadaSimple<E> divide() {...}

    ...
}
```