PRUEBA EVALUABLE ESTRUCTURAS DE DATOS- PARTE I

1. Explicar por qué la complejidad algorítmica se mide en órdenes (0.5 puntos).

2. Calcular la complejidad algorítmica del algoritmo recursivo tipo divide y vencerás de la Figura 1, asumiendo que el árbol que se pasa como parámetro de la función es un árbol binario de búsqueda balanceado. La complejidad de las operaciones *esHoja()*, *getHijoIzquierdo()*, *getHijoDerecho()*, *setHijoIzquierdo()* y *setHijoDerecho()* es O(1) (0.5 puntos).

```
PROC imagenEspecular(REF a : ArbolBinarioBusqueda(E)) ES
    SI (NOT a.esHoja()) ENTONCES
       imagenEspecular(a.getHijoIzquierdo());
       imagenEspecular(a.getHijoDerecho());
       aux : ArbolBinarioBusqueda(E)
       aux := a.getHijoIzquierdo();
       aux.setHijoIzquierdo(a.getHijoDerecho());
       aux.setHijoDerecho(a.getHijoIzquierdo());
    FINSI
FINPROC
```

Figura 1. Algoritmo recursivo tipo divide y vencerás *imagenEspecular*

3. Dada la implementación de un árbol binario de búsqueda mostrada en la Figura 2, implementar el método *esAscendiente()* de acuerdo con la especificación proporcionada en su comentario de documentación. La eficiencia del método debería ser *O(log n)*, asumiendo que el árbol está balanceado. Justificar que el método tiene la eficiencia pedida. (1 punto)

```
public class ArbolBinCE <E extends Comparable<E>> {

   private class Nudo<E> {
        Nudo<E> padre, hijoIzq, hijoDer;
        E contenido;

        Nudo(E e) { contenido = e; }
   } // Nudo

   private Nudo<E> raiz;
   /**
    * Permite conocer si el elemento eleAscen ocupa un nodo en el árbol
    * que es un ascendiente del nodo que ocupa eleDescen
    * @param eleDescen elemento que podría ocupar un nodo descendiente de eleAscen
    * @param eleAscen elemento que podría ocupar un nodo ascendiente de eleDesce
    * @return verdadero si eleAscen es un ascendiente de eleDescen y falso en caso
    * contrario (también falso el caso en que uno de los elementos no esté en el
árbol) */
   public boolean esAscendiente(E eleDescen, E eleAscen) {
   ... implementar por el alumno ...
   } // esAscendiente
 } // ArbolBinCE
```

Figura 2. Clase Árbol Binario de Búsqueda

4. Usando la implementación de una de una lista doblemente enlazada que utiliza nodos de cabecera y cola tal como la que se muestra en la Figura 3, implementar el método *mueveAlFinal()* de acuerdo a la funcionalidad descrita en su comentario de documentación. La eficiencia del método deberá ser *O(n)* (donde *n* es el número de elementos en la lista). Deberá implementarse como un método independiente que no llame a ningún otro método de la clase (No se pueden usar métodos como *add*, *remove* o *size*) (1 punto).

```
public class ListaDobleEnlaceNCC<E> {

    private Celda<E> principio;
    private Celda<E> fin;

    private class Celda<E> {
        E contenido;
        Celda<E> siguiente;
        Celda<E> anterior;
        Celda(E cont) { contenido=cont; } // Celda
    } // Celda

    public ListaDobleEnlaceNCC() {
        // crea las celdas de cabecera y cola
        Principio = new Celda<E>(null);
        fin       = new Celda<E>(null);
    } // ListaDobleEnlaceNCC

    /**
     * Mueve al final de la lista todos los elementos iguales al
     * elemento que recibe como parámetro
     * @param e los elementos iguales a este se mueven al final
     */
    public void mueveAlFinal(E e) {
              ... implementar por el alumno ...
    } // mueveAlFinal
} // ListaDobleEnlaceNCC
```

Figura 3. Implementación de listas enlazadas con nodos de cabecera y cola

5. En un intento de limitar el tamaño de las listas de una tabla de dispersión abierta se ha optado por combinar la dispersión abierta con una estrategia de doble dispersión. Por tratarse de una estrategia de doble dispersión, las llaves deben contar con dos funciones de dispersión independientes, por lo que implementan la interfaz *DobleDispersión*, mostrada en la Figura 4.

```
public interface IDobleDispersion {
   public int hashCode1();
   public int hahsCode2();
} // IDobleDispersion
```

Figura 4. Interfaz *DobleDispersión*

Sea *SizeMax* el tamaño máximo al que se intenta limitar las listas. Mientras que el tamaño de las listas está por debajo de *SizeMax*, la tabla se comporta como una tabla de dispersión abierta normal que utiliza *hashCode1()* como función de dispersión. Si al añadir una nueva asociación llave-valor se encuentra que la lista obtenida utilizando *hashCode1()* ya ha alcanzado el tamaño *SizeMax*, se utiliza la segunda función de dispersión para localizar otra lista utilizando la expresión *(hashCode1+hashCode2)%tamaño_tabla*. La nueva asociación se añade en esta segunda lista independientemente de su tamaño.

Por ejemplo en la Figura 5 se muestra la inserción de una pareja llave-valor. Al no poder insertarse en la en la lista 5 (puesto que esta lista ya ha alcanzado el tamaño máximo), la inserción finalmente se realiza en la lista 2.

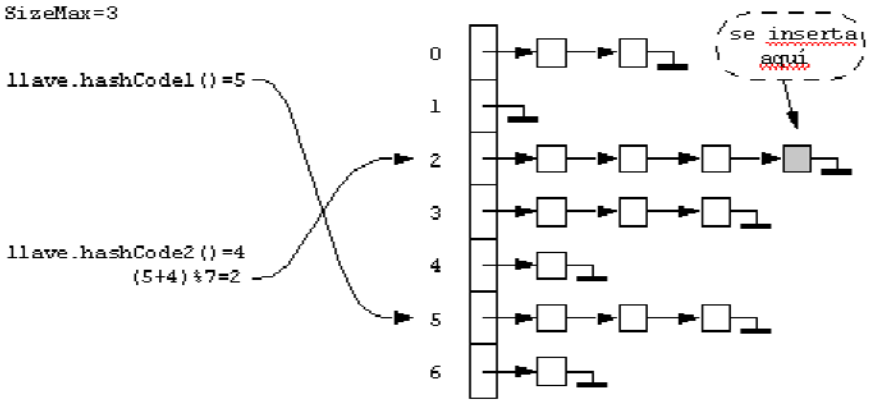

Figura 5. Esquema de doble dispersión propuesto

Tomando como punto de partida el código que aparece en la Figura 5, implementar los métodos *put()* y *remove()*. Sería deseable que únicamente se explorara la lista obtenida mediante la segunda dispersión en aquellos casos que fuera estrictamente necesario (25% de la nota asignada a la cuestión). Para lograr este propósito podría ser necesario añadir algún atributo a la clase *Entry* (2 puntos).

```
public class TablaDispersion<K extends IDobleDispersion, V> {

   private List<Entry<K,V>>[] tabla;

   private class Entry<K,V> {
      private K llave;
      private V valor;

      public Entry(K llave, V valor) {
        this.llave=llave;
        this.valor=valor;
      } // Entry
   } // Entry

   public TablaDispersion(int tamano) {
      tabla =  new LinkedList[tamano];
      for (int i = 0; i < tamano; i++) {
          tabla[i] = new LinkedList<Entry<K,V>>();
      } // for
   } // TablaDispersion

   public V put(K llave, V valor) {
       implementar por el alumno  }

   public V remove(K clave) {
       implementar por el alumno  }
} // TablaDispersion
```

Figura 6. Clase *TablaDispersion*

Aclaraciones (aplicables Parte I y Parte II)

(1) Las implementaciones se pueden realizar tanto en pseudocódigo como en sintaxis Java. En caso de optar por usar Java, no se permite en ningún caso la importación y uso de las bibliotecas o clases que se proporcionan de forma predefinida con el lenguaje en la Parte I del examen.

(2) En todos los casos se valorará principalmente la claridad y corrección técnica de los razonamientos realizados por el alumno por encima de los resultados concretos finales obtenidos.

(3) Se valorará especialmente el uso de un lenguaje técnico adecuado así como la claridad, precisión y capacidad de síntesis de las respuestas. Se recomienda evitar el uso de un lenguaje coloquial, abreviaturas o expresiones más propias de los mensajes cortos enviados a través de teléfonos móviles.

(4) La duración de cada parte será de 2 horas.

PRUEBA EVALUABLE ESTRUCTURAS DE DATOS- PARTE II

6. Dada la especificación algebraica del TAD *Desconocido* del Apéndice A (1.5 puntos):
   a. Indicar si el TAD representado admite elementos repetidos. Justificar la respuesta (0.25 puntos).
   b. Indicar si el orden de los elementos es relevante en el TAD representado. Justificar la respuesta (0.25 puntos).
   c. Indicar si se puede acceder a cualquier elemento almacenado en dicho TAD o por el contrario existen elementos destacados. Justificar la respuesta (0.25 puntos).
   d. Implementar la operación *paFuera* usando como soporte para la implementación un array. Indicar los atributos que formarían parte de la clase especificando con claridad los invariantes, si los hubiere (0.75 puntos).

7. Sistema de Gestión de la Constitución de Cádiz de 1812, alias "*La Pepa*" (3.5 puntos).

Cádiz 1812. Todo el territorio nacional se encuentra ocupado por las tropas francesas lideradas por José Bonaparte, alias Pepe Botella, a excepción de la ciudad de Cádiz; la cual resiste de forma heroica al invasor. Mientras las gaditanas se hacen tirabuzones con las bombas francesas, un grupo de parlamentarios se afanan en producir el primer texto constitucionalista de la Historia de España. Para ello precisan de cierto soporte informático, el cual, por razones que los historiadores no alcanzan a comprender, existía en aquella época.

La constitución se compondrá de una serie de *títulos*. Cada título contiene un número y un nombre, que es único para cada título. Cada título se compone a su vez de una serie de *capítulos*, que al igual que los títulos, tienen un número y un nombre, que es único para cada capítulo. El número de un capítulo sólo es único dentro de un título, pudiendo existir capítulos con los mismos números pero pertenecientes a diferentes títulos. A su vez, cada capítulo tiene una serie de *artículos*, que también tienen un número y un nombre. El número de un artículo sólo es único para cada artículo dentro de un capítulo, pudiendo existir artículos con los mismos números pero pertenecientes a diferentes capítulos.

Cada vez que se da de alta un nuevo título, capítulo o artículo el sistema debe comprobar que no exista otro título, capítulo o artículo con el mismo nombre, respectivamente. El número que corresponde a cada título, capítulo o artículo los debe asignar el sistema automáticamente al darlos de alta.

Además, cada vez que se redacta un nuevo artículo, los parlamentarios definen una serie de palabras clave para facilitar su búsqueda. Cada artículo puede tener asociada varias palabras claves. Una palabra clave puede ser compartida por diversos artículos.

Por ejemplo, el artículo 12 del capítulo 2 del título 2 establece: "*La religión de la Nación española es y será perpetuamente la católica, apostólica, romana, única verdadera. La Nación la protege por leyes sabias y justas, y prohíbe el ejercicio de cualquiera otra*." Por tanto, los parlamentarios podrían definir como palabras clave: "fe de la nación", "religión" o "*asuntos de orden nacional*".

La búsqueda de los títulos, capítulos y artículos se realiza principalmente por nombre. Los capítulos una vez insertados, nunca se borran. Los números de los títulos, capítulos y artículos se asignan de forma creciente y consecutiva. Es decir, si el sistema tiene introducidos 6 títulos, al introducir uno nuevo, se le asigna como número 7.

También es frecuente que los parlamentarios usen el sistema para ver la lista de artículos que están asociados a una cierta palabra clave, la cual deberá haber sido dada de alta previamente en el sistema.

Con cierta asiduidad, los coordinadores de la elaboración del texto constitucional obtienen listados donde deben aparecer los diferentes títulos ordenados por número. Dentro de cada título, deben aparecer los capítulos ordenados por número, y dentro de cada capítulo, los artículos, también ordenados por número, con su correspondiente texto.

Con la información proporcionada, decidir:

(1) Qué TADs son los más adecuados para almacenar la información necesaria para elaborar la primera carta magna del estado español de forma que se optimicen las operaciones más comunes (0.5 puntos).
(2) Qué implementación resulta la más adecuada para cada TAD seleccionado en el apartado anterior (0.5 puntos).
(3) Si al presidente de la comisión constitucional le apeteciese almacenar las palabras claves en un árbol binario de búsqueda autobalanceado, ¿qué tipo de árbol le recomendaría? (0.25 puntos)
(4) Crear una clase *GestorConstitucion*, indicando sólo los atributos que posee y los invariantes que poseen dichos atributos (0.5 puntos).
(5) Crear las clases *Título, Capítulo* y *Artículo* indicando sólo sus atributos y qué restricciones deberían cumplir tales clases para poder ser almacenadas en los TADs elegidos en el apartado (1) usando las implementaciones seleccionadas en el apartado (2). Indicar, usando lenguaje natural, qué modificaciones se tendrían que realizar sobre la implementación del apartado (2) para que las clases satisfagan las restricciones indicadas (0.25 puntos).
(6) Implementar el método *listarConstitucion()*, el cual muestra el estado actual de la constitución por pantalla (0.25 puntos).
(7) Implementar el método *altaCapitulo(nombreCapitulo: String, titulo : String) : boolean*, que sirve para dar de alta un nuevo capítulo en el texto constitucional. *nombreCapitulo* es el nombre del capítulo que se desea insertar y *titulo* el nombre del título al cual pertenece el capítulo. La función devuelve *true* si se ha podido insertar el capítulo y *false* en el caso contrario (0.5 puntos).
(8) Implementar el método *listarPorPalabraClave(palabra: String)*, que sirve para mostrar, en cualquier orden, todos los capítulos asociados a una cierta palabra clave (0.25 puntos).
(9) Sabiendo que la constitución tiene *n* títulos, *m* capítulos y *p* artículos; el número máximo de artículos por capítulo es *O(sin(n))* y el número máximo de capítulos por título es *O(log(n*p));* calcular la complejidad de las operaciones implementadas en los apartados (6), (7) y (8) (0.5 puntos).

Apéndice A. Especificación Algebraica del TAD Desconocido

```
espec Desconocido
   usa Boolean, Natural
   generos Desconocido (Element)como Desc
   parametro
     generos Element como E
     operaciones
        igual : Element Element -> Boolean
     variables
        x, y, z : Element;
     ecuaciones
        igual(x,x) = TRUE
        igual(x,y) = igual(y,x)
        [igual(x,y) and igual(y,z)] igual(x,z) = TRUE;
   fparametro

operaciones
   emptyDesc  :  -> Desc
   entra : Desc E -> Desc
   parcial consulta : Desc -> E
   parcial paFuera  : Desc -> Desc
   esVacia : Desc -> Boolean
vars
   s : Desc; x, y : Element;
precondiciones
   [esVacia (s) != TRUE] paFuera(s), consulta(s)
ecuaciones
    // Las generadoras son emptyDesc y entra(s,x) y son libres
    // Los patrones del tipo son:
    //   - emptyDesc
    //   - entra(s,x)
    paFuera(entra(emptyDesc,x)) = emptyDesc
    [igual(x,y) == TRUE] paFuera(entra(entra(s,x),y)) == paFuera(entra(s,x))
    [igual(x,y) != TRUE] paFuera(entra(entra(s,x),y)) == entra(s,x)
    consulta(entra(s,x)) = x
    esVacia (emptyDesc)  = TRUE
    esVacia (entra(s,x)) = FALSE
fespec
```

Mario Aldea Rivas
Pablo Sánchez Barreiro