PRUEBA EVALUABLE ESTRUCTURAS DE DATOS- PARTE I

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

Un orden representa el conjunto de funciones que poseen un mismo ritmo de crecimiento. A la hora de comparar dos algoritmos, lo que nos interesa es comparar sus ritmos de crecimiento, es decir, la capacidad de enfrentar cada algoritmo a problemas mayores. Por ejemplo, si tenemos un algoritmo de complejidad O(n), al aumentar el tamaño del problema sabemos que se producirá un aumento en el tiempo de ejecución directamente proporcional al incremente realizado. Si la complejidad fuese $O(2^n)$, al incrementar el tamaño del problema, el aumento en tiempo de ejecución ya no es directamente proporcional al tiempo de ejecución, pudiendo existir un tamaño máximo del problema para el cual el tiempo de ejecución se dispare hasta valores inmanejables en la práctica. Por tanto, un algoritmo O(n) ofrece una mayor capacidad o resistencia para tratar con tamaños de problemas grandes que uno de complejidad $O(2^n)$. Dicha capacidad queda reflejada en el orden de la función que define el tiempo de ejecución de un algoritmo.

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*

Aplicando el teorema maestro:
a = 2 (contando simplemente)
b = 2 (por un árbol binario balanceado y quedarnos en cada llamada recursiva con una rama)
k = 0, por ser tanto el caso base como la parte iterativa del caso recursivo O(1).

Por tanto, aplicando el teorema maestro, $O(n^{\log_2(2)}) = O(n)$

PRUEBA EVALUABLE ESTRUCTURAS DE DATOS- PARTE II

3. 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).

El TAD admite elementos repetidos ya que no hay ninguna ecuación entre generadoras (impurificadora) que indique que los términos canónicos entra(entra(emptyDesc,x),x) y entra(emptyDesc,x) sean iguales, por lo que hemos de considerarlos como términos canónicos distintos.

b. Indicar si el orden de los elementos es relevante en el TAD representado. Justificar la respuesta (0.25 puntos).

El orden es relevante ya que no hay ninguna ecuación entre generadoras (impurificadora) que indique que los términos canónicos entra(entra(emptyDesc,x),y) y entra(entra(emptyDesc,y),x) sean iguales, por lo que hemos de considerarlos como términos canónicos distintos, siendo la posición u orden de los elementos en la estructura relevantes.

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).

Dado el comportamiento de la operación *consulta*, deducimos que solo podemos acceder al último elemento almacenado, poseyendo el TAD un comportaiento similar, aunque no idéntico al de una pila con política de acceso LIFO.

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).

```
CLASE Desc(E) ES
      // NOTA: Los objetos de la clase E deben ser comparables por igualdad
      MAX_VECTOR ES NATURAL CON VALOR 1000;
      elementos : VECTOR MAX_VECTOR DE E;
      // Inv: forall i : NATURAL, 0 <= I < ultimo; elementos[i]  != NULO;
      ultimo : NATURAL;

      // Pre:  NOT this.esVacia()  // Versión recursiva
      FUNCION paFuera()  : Desc(E) ES
        SI (ultimo == 1) ENTONCES
             ultimo = 0;
        SINOSI (elementos[ultimo-1].igual(elementos[ultimo-2])) ENTONCES
              ultimo := ultimo -1;
              this.paFuera()
        SINO
              ultimo := ultimo -1;
        FINSI
        DEVOLVER esteObj;
      FINFUNCION
   // Pre: NOT this.esVacia()  // Versión iterativa
```

```
    FUNCION paFuera()  : Desc(E) ES
      MIENTRAS (ultimo != 1)  AND
               (elementos[ultimo-1].igual(elementos[ultimo-2])) HACER
        ultimo := ultimo -1;
      FINMIENTRAS
      ultimo := ultimo -1;
      DEVOLVER esteObj;
     FINFUNCION
FINCLASE
```

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

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) Crear una clase *GestorConstitucion*, indicando sólo los atributos que posee y los invariantes que poseen dichos atributos (0.5 puntos).

(4) 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).

```
CLASE GestorConstitución ES

    titulos : LISTA<Titulo>;
    // Inv: titulos != NULO
    // Inv: forall t in titulos, t != NULO
    // Inv: forall t in titulos, titulos[t.getNumber()-1].igual(t)
    // Explicación: Como número para cada título utilizamos su posición en la
    // lista de títulos más uno
    // Inv: forall t1, t2 in titulos;
    // (NOT (t1.getNumber() == t2.getNumber(t2)) IMPLIES
    //                              (NOT t1.getNombre().igual(t2.getNombre()))
    mapaTitulos : MAPA<CadenaCaracteres,Titulo>;
    // Inv: mapaTitulos != NULO
    // Inv: forall s : CadenaCaracteres,
    //          mapaTitulos.containsKey(s) IMPLIES mapaTitulos.get(s) != NULO
    // Inv: forall s : CadenaCaracteres,
    //          mapaTitulos.containsKey(s) IMPLIES
    //                  mapaTitulos.get(s).getNombre().igual(s)
    // Inv: forall s : CadenaCaracteres,
    //          mapaTitulos.containsKey(s) IMPLIES
    //                  titulos.contains(mapaTitulos.get(s))
```

```
    mapaArticulos : MAPA<CadenaCaracteres,Articulo>;
    // Inv: mapaArticulos != NULO
    // Inv: forall s : CadenaCaracteres,
    //       mapaArticulos.containsKey(s) IMPLIES mapaArticulos.get(s) != NULO
    // Inv: forall s : CadenaCaracteres,
    //       mapaArticulos.containsKey(s) IMPLIES
    //                 mapaArticulos.get(s).getNombre().igual(s)
    // Inv: forall s : CadenaCaracteres,
    //       mapaArticulos.containsKey(s) IMPLIES
    //             mapaCapitulos.containsKey( mapaArticulos.get(s).
    //                                         getArticulo().getNombre())

    mapaCapitulos : MAPA<CadenaCaracteres,Titulo>;
    // Inv: mapaTitulos != NULO
    // Inv: forall s : CadenaCaracteres,
    //       mapaCapitulos.containsKey(s) IMPLIES mapaCapitulos.get(s) != NULO
    // Inv: forall s : CadenaCaracteres,
    //          mapaCapitulos.containsKey(s) IMPLIES
    //                 mapaCapitulos.get(s).getNombre().igual(s)
    // Inv: forall s : CadenaCaracteres,
    //       mapaCapitulos.containsKey(s) IMPLIES
    //             mapaTitulos.containsKey(mapaCapitulos.get(s).
    //                                         getTitulo().getNombre())

    palabrasClave : CONJUNTO<CadenaCaracteres>;
    // Inv: palabrasClave != NULO

    mapaPalabraArticulo : MAPA<CadenaCaracteres,Articulo>
    // Inv: forall s : CadenaCaracteres,
    //          mapaPalabraArticulo.containsKey(s) IMPLIES
    //                 mapaPalabraArticulo.get(s)!= NULO;
    // Inv: Por cada artículo perteneciente al conjunto de artículos
    //      asociado a una palabra clave, dicho artículo es un artículo
    //      de la constitución, es decir, su título está en la lista de
    //      títulos y está dado de alta en el mapa de artículos
FINCLASE

CLASE Titulo ES
    capitulos : LISTA<Capitulo>;
    // Inv: capitulos != NULO
    // Inv: forall c in capitulos, c != NULO
    // Inv: forall c in capitulos, capitulos[c.getNumber()].igual(c)
    // Explicación: Como número para cada capítulo utilizamos su posición
    // en la lista
    // Inv: forall c1, c2 in capitulos;
    // (NOT c1.getNumber() != c2.getNumber(t2)) IMPLIES
    //                             (NOT c1.getNombre().igual(c2.getNombre()))
    nombre : CadenaCaracteres;
    // Inv: nombre != NULO
    // Inv: forall t1, t2: Titulo;
    //             t1.getNombre().igual(t2.getNombre()) iff t1.igual(t2)
    numero : NATURAL
FINCLASE
```

```
CLASE Capitulo ES
    articulos : LISTA<Articulo>;
    // Inv: articulos != NULO
    // Inv: forall a in articulos, a != NULO
    // Inv: forall a in articulos, articulos.get(a.getNumber()).igual(a)
    // Explicación: Como número para cada articulo utilizamos su posición
    // en la lista
    // Inv: forall a1, a2 in articulos;
    // (NOT a1.getNumber() != a2.getNumber(t2)) IMPLIES
    //                              (NOT t1.getNombre().igual(t2.getNombre()))
    nombre : CadenaCaracteres;
    // Inv: nombre != NULO
    // Inv: forall t1, t2: Titulo;
    //             t1.getNombre().igual(t2.getNombre()) iff t1.igual(t2)
    numero : NATURAL
    titulo : Titulo;
    // Inv: titulo.getCapítulos.contains(this);
FINCLASE

CLASE Articulo ES
    nombre : CadenaCaracteres;
    // Inv: nombre != NULO
    // Inv: forall t1, t2: Titulo;
    //             t1.getNombre().igual(t2.getNombre()) iff t1.igual(t2)
    numero : NATURAL
    texto : CadenaCaracteres;
    // Inv: texto != NULO
    palabrasClave : CONJUNTO<CadenaCaracteres>;
    // Inv: palabrasClave != NULO;
    capitulo : Capitulo;
    // Inv: capitulo.getArticulos.contains(this);
FINCLASE
```

Justificaciones:

Dado que el orden de los títulos, capítulos y artículos es relevante para poder imprimir la constitución de forma ordenada, usaremos colecciones con orden relevante, es decir Listas o Listas sin repeticiones. Además definiremos mapas para acceder a los diferentes elementos por nombre. Por tanto, dado que podemos usar el mapa de forma natural para controlar las repeticiones, podemos usar listas en lugar de listas sin repetición.

Usaremos conjunto para almacenar las palabras clave (su orden es relevante y no queremos almacenar elementos repetidos). Además, definiremos un mapa para recuperar el conjunto de artículos asociados con una misma palabra clave.

Para las listas podemos usar listas enlazadas simples o vectores con un tamaño máximo adecuado. Dado que los elementos no se borran, los vectores no poseen ningún inconveniente y además ofrecen la posibilidad de mantener los elementos contiguos en memoria, lo cual seguro que era importante dadas las limitaciones del hardware de la época. Destacar que además esta estructura no se utilizará para realizar búsquedas, sino para recorrerla linealmente a la hora de imprimir la constitución.

Para los mapas no hay nada que indique que no podamos usar tablas de dispersión, las cuales poseen una mayor eficiencia que los árboles autobalanceados.

Para el conjunto de palabras clave y el conjunto de artículos asociado a una palabra clave podemos utilizar igualmente una tabla de dispersión, por las mismas razones anteriores.

Por tanto, la clase Artículo debe ser dispersable y comparable por igualdad, ya que se almacenan en conjuntos implementados sobre árboles de dispersión.

El resto de las clases no precisan de satisfacer ninguna restricción adicional, ya que aunque se almacenan en mapas sobre tablas de dispersión, lo que ha de ser dispersable y comparables por igualada son sus claves.

(5) 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)

Le recomendaría un árbol AVL por tener menos altura que el rojinegro (O(1.44log(n)) frente a O(2log(n))) y por tanto permitir unas búsquedas más eficientes, ya que éstas parecen ser las principales operaciones que se ejecutarán sobre este mapa.

(6) Implementar el método *listarConstitucion()*, el cual muestra el estado actual de la constitución por pantalla (0.25 puntos).

```
PROC listarConstitucion() ES
   PARA i:NATURAL DESDE 0 HASTA titulos.size() HACER
      titulo : Titulo;
      titulo := titulos.get(i); // O(1) al ser lista sobre vector
      IMPRIMIR titulo.getNumero() + “.” + titulo.getNombre(); // O(1)
      PARA j:NATURAL DESDE 0 HASTA titulo.getCapitulos().size() HACER // O(1)
         capitulo : Capitulo;
         capitulo := titulo.getCapitulos().get(j); // O(1) vector
         IMPRIMIR capitulo.getNumero() + “.” + capitulo.getNombre(); // O(1)
         // O(1)
         PARA k:NATURAL DESDE 0 HASTA capitulo.getArticulos().size() HACER
            articulo : Articulo;
            articulo := articulo.getArticulos().get(k); // O(1)
             // O(1)
            IMPRIMIR articulo.getNumero() + “.” + capitulo.getNombre();
            IMPRIMIR articulo.getTexto(); // O(1)
         FINPARA
      FINPARA
   FINPARA
FINPROC
```

(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).

```
// Pre: mapaTitulos.containsKey(titulo) AND (nombreCapitulo != NULO)
FUNCION altaCapitulo(nombreCapitulo: CadenaCarateres,
                     titulo : CadenaCarateres) : BOOLEANO ES

   result : BOOLEANO;
   result := FALSO;
   SI NOT mapaCapitulos.containsKey(nombreCapitulo) ENTONCES // O(1)
      result := VERDADERO;
      numero : NATURAL;
      numero := mapaTitulos.get(titulo).getArticulos().size()+1; // O(1)
      capitulo : Capitulo;
      capitulo := NUEVO Capitulo(nombreCapitulo,numero,titulo); // O(1)
      mapaTitulos.get(titulo).getArticulos().add(capitulo); // O(1)
      mapaCapitulos.put(nombreCapitulo,capitulo) // O(1)
   FINSI
   DEVOLVER result;
FINFUNCION
```

(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).

Solución 1: Los capítulos no tienen asociadas palabras claves, por lo que esta función no tiene sentido.

Solución 2: Entendemos que donde dice capítulo quiere decir artículo.

```
// Pre: mapaPalabraArticulo.containsKey(palabra)
PROC listarPorPalabraClave(palabra: CadenaCaracteres) ES
     PARA CADA a : Articulo EN mapaPalabraArticulo.get(palabra) HACER
          IMPRIMIR a;
     FINPARA
FINPROC
```

Solución 3: Entendemos que un capítulo está asociado a una palabra clave si tiene al menos un artículo asociado a esa palabra clave.

```
// Pre: mapaPalabraArticulo.containsKey(palabra)
PROC listarPorPalabraClave(palabra: CadenaCaracteres) ES
     capitulos : CONJUNTO<Capitulo>;
     capitulos  := NUEVO CONJUNTO_HASH<Capitulo>(…);

     PARA CADA a : Articulo EN mapaPalabraArticulo.get(palabra) HACER
          capitulos.add(a.capitulo)
     FINPARA

     IMPRIMIR capitulos
FINPROC
```

(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).

Apartado 6, primer bucle n iteraciones, por O(log(n*p)) iteraciones en el segundo lazo y O(sin(n)) el tercer lazo. El resto de las operaciones son O(1), por lo que la complejidad será O(n)*O(log(n*p))* O(sin(n)) = O(n* log(n*p)* sin(n)).

Apartado 7. O(1)
Apartado 8.
Solución 1. No aplica
Solución 2. O(n), donde n es el número máximo de artículos asociados a una palabra clave.
Solución 3. O(n), donde n es el número máximo de artículos asociados a una palabra clave.

Pablo Sánchez Barreiro