ESTRUCTURA DE DATOS Y ALGORITMOS **1-Septiembre-2005**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

1. Ejercicio (3,5 puntos)

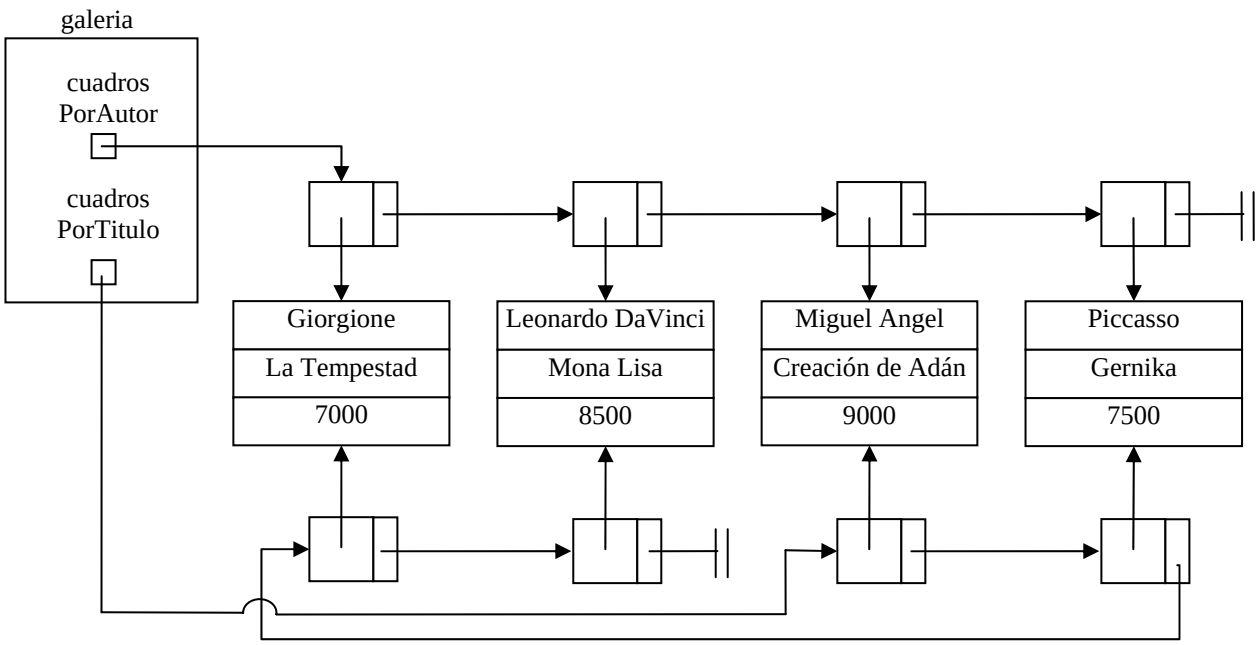
a) Complentar la especificación

```
public class Galeria  {
   LinkedList cuadrosPorTitulo;
   LinkedList cuadrosPorAutor;
   LinkedListItr itrCuadrosPorTitulo;
   LinkedListItr itrCuadrosPorAutor;
}

class Node {
   Object element;
   Node next;
}
public class LinkedList{
   Node top;
}
```

```
public class LinkedListItr {
   LinkedList theList;
   Node current;
   Node preceding;
}
```

b) Representación de la información en listas:



c) Especificación de los métodos que se van a utilizar

```
public void advance();
// avanza en la lista con la variable current, pasando al siguiente nodo del nodo current
public boolean isOnList();
// verifica si la variable current referencia un nodo de la lista, en caso afirmativo devuelve true  y en
// caso contrario false
public void goFirst();
 //actualiza la variable current, el cuál referencia el primer nodo de la lista
public Object getCurrent();
// devuelve el objeto que esta referenciado en el nodo, el cuál es accesible mediante la variable current
```

```
public void insert(Object obj);
// inserta la información obj en la lista, creando un nuevo nodo después del nodo referenciado por la
// variable current. Finalmente, current referencia el nuevo nodo creado.
public void insertFirst(Object obj);
// inserta la información en la primera posición de la lista, creando un nuevo nodo con la información
// obj. Finalmente, current referencia el nuevo nodo creado.
```

Implementación:

```
  public void añadir(CuadroPintura  cuadro){
    CuadroPintura cAntT = null;
    CuadroPintura cAntA = null;
    boolean enc = false;
    CuadroPintura c = null;

    //buscar posición en la lista por titulo, NO puede haber repetidos
    itrCuadrosPorTitulo.goFirst();
    while ( (itrCuadrosPorTitulo.isOnList()) && !enc ) {
      c = (CuadroPintura)itrCuadrosPorTitulo.getCurrent();
      if (c.getTitulo().compareTo(cuadro.getTitulo())<0) {
        cAntT = c;
        itrCuadrosPorTitulo.advance();
      }
      else enc = true;
    }

    if ((enc) && (c.getTitulo().compareTo(cuadro.getTitulo())==0))
      System.out.println("Error, el cuadro con el titulo " + cuadro.getTitulo() +
                          "  ya esta");
    else {
      enc = false;

      //buscar posición en la lista por autores, puede haber repetidos y están
      //ordenados por titulo
      itrCuadrosPorAutor.goFirst();
      while ( (itrCuadrosPorAutor.isOnList()) && !enc ) {
        c = (CuadroPintura)itrCuadrosPorAutor.getCurrent();
        if ((c.getAutor().compareTo(cuadro.getAutor())<0) ||
            ((c.getAutor().compareTo(cuadro.getAutor())==0) &&
             (c.getTitulo().compareTo(cuadro.getTitulo())<0))){
          cAntA = c;
          itrCuadrosPorAutor.advance();
        }
        else enc = true;
      }

      // insertar el cuadro en la lista por titulo
      if (cAntT == null)
        itrCuadrosPorTitulo.insertFirst(cuadro);
      else  {
        itrCuadrosPorTitulo.find(cAntT);
        itrCuadrosPorTitulo.insert(cuadro);
      }
      // insertar el cuadro en la lista por autor
      if (cAntA == null)
        itrCuadrosPorAutor.insertFirst(cuadro);
      else {
        itrCuadrosPorAutor.find(cAntA);
        itrCuadrosPorAutor.insert(cuadro);
      }
    }
  }
```

2. Ejercicio. (2,5 puntos)

1) Especificación/Parametrización

Entrada: dos árboles de enteros, *arbol1* y *arbol2*
Salida: un boolean, *true* si *arbol1* es prefijo de *arbol2* y *false*, en caso contrario

public static boolean esPrefijo(BinTreeItr arbol1, BinTreeItr arbol2)

A la hora de tratar el árbol utilizaré la definición recursiva del árbol, ya que esto me ayuda a simplificar el problema. Siguiendo la definición de esPrefijo, podemos decir que un subárbol binario **a1** es prefijo de otro subárbol binario **a2**, cuando **a1** coincide con la parte inicial del subárbol **a2** tanto en el contenido de los elementos como en su situación.
Por lo tanto, para identificar un subárbol necesitaré utilizar el tipo BTNode y para resolver el subproblema un método auxiliar, el cuál me ayudará a resolver el problema inicial. A continuación detallo la especificación/parametrización del subproblema

Entrada: dos subárboles de enteros, *a1* y *a2*
Salida: un boolean, *true* si *a1* es prefijo de *a2* y *false*, en caso contrario.

private static boolean esPrefijo(BTNode a1, BTNode a2)

2) Diseño

2.1) Casos triviales:

- **Si** a1 es vacio → **devolver** true
- **Si** a1 no es vacio y a2 es vacio → **devolver** false
- **Si** a1 y a2 no son vacios y las raices son distintas → **devolver** false

2.2) Casos generales:

- **Si** a1 y a2 no son vacios y las raices son iguales →
  devolver esPrefijo(a1.left, a2.left) && esPrefijo(a1.right, a2.right)

3) Implementación

```
public static boolean esPrefijo(BinTreeItr arbol1, BinTreeItr arbol2) {
  return esPrefijo(arbol1.bTree.root, arbol2.bTree.root);
}

private static boolean esPrefijo(BTNode a1, BTNode a2){
  if (a1 == null) return true;
  else if (a2 == null) return false;
  else{
    int raiz1 = ((Integer) a1.content).intValue();
    int raiz2 = ((Integer) a2.content).intValue();
    if (raiz1 != raiz2) return false;
    else return (esPrefijo(a1.left, a2.left) &&
                 esPrefijo(a1.right, a2.right));
  }
}
```

3. Ejercicio (4 puntos)

a) Definición de las estructuras

```
public class Libro  {
  private String titulo;
  private int peso;
  private int grosor;
}
```

```
public class Caja  {
  public static int NUM_MONTONES_MAX = 4;
  public static int ALTURA_MAX = 50;
  MontonLibros[] montones =
      new MontonLibros[NUM_MONTONES_MAX];
}
```

```
public class MontonLibros  {
  private Stack libros = new Stack();
      //como los libros se suelen apilar uno encima de otro,
      //la estructura más lógica es utilizar una pila
  private int altura = 0;
      //para realizar de modo eficiente los dos métodos que nos piden es
      //interesante mantener cuál es la  altura del montón. Esta variable se
      //actualizará cuando se ponen y quitan libros del montón.
```

```
public class BibliotecaEmbalada  {
  private Vector cajas = new Vector(5);
}
Otra opción:
public class BibliotecaEmbalada  {
  private Caja[] cajas = new Caja[5];
}
```

b) **Diseñar e implementar** un método que extraiga los libros que contiene una caja:

1) **Diseño**

Especificación

Entrada: una caja de tipo Caja
Salida: un entero, el peso de la caja
un entero, la altura del montón que más libros contiene

Se necesita definir una clase para agrupar los dos valores de salida. La clase se llamará *PesoYAltura*.

Algoritmo **extraerLibros**(Caja caja)

```
peso = 0;
numLibrosMax = Integer.MinValue;
alturaMax = 0;
mientras haya montones en la caja
      monton = recuperarMontonDe(caja);
      Si numeroDeLibrosEn(monton) > numLibrosMax entonces
            alturaMax = alturaDe(monton);
      finSi
      mientras no EstaVacio(monton)
            libro = quitarLibroDel(monton);
            peso += pesoDel(libro);
      finMientras
finMientras
devolver PesoYAltura(peso, alturaMax);
```

2) **Implementación:** Dos opciones:
   i. El método es de la clase Caja, lo cuál NO se necesita pasar la caja como parámetro
   ii. El método es independiente, lo cuál la caja se pasa como parámetro

En este caso lo resolvemos implementando la opción i):

```
public PesoYAltura extraerLibros(){
  int peso = 0;
  int numLibrosMax = Integer.MIN_VALUE;
  int alturaMax = 0;
  for(int i=0;i < NUM_MONTONES_MAX; i++){
    MontonLibros monton = montones[i];
    if (monton.getNumLibros() > numLibrosMax)
      alturaMax = monton.getAltura();
    while (!monton.estaVacio()){
      Libro libro = monton.quitar();
      peso += libro.getPeso();
    }
  }
  return new PesoYAltura(peso, alturaMax);
}

public class MontonLibros {
  public int getNumLibros(){
    return libros.size();
  }
  public int getAltura(){
    return altura;
  }
  public boolean estaVacio(){
    return libros.isEmpty();
  }
  public Libro quitar(){
    Libro libro = null;
    if (!libros.isEmpty()){
      libro = (Libro) libros.pop();
      altura -= libro.getGrosor();
      numLibros --;
    }
    return libro;
  }
}

public class Libro {
  public int getPeso(){
    return peso;
  }
}

public class PesoYAltura  {
  private int peso;
  private int altura;
}
```

c)

i. **Definir** la estructura de datos más adecuada para implementar la estantería.

```
public class Estanteria  {
  private Stack libros;
}
```

ii. **Implementar** un método que embale, si puede, todos los libros de la estantería en una caja. Intentar llenar la caja lo máximo posible y dejar en la estantería aquellos libros que no entran, en el mismo orden inicial.

Se intenta llenar al máximo la caja, con las siguiente decisiones tomadas:
- mientras haya libros en la estantería y haya sitio en la caja, se intenta introducir un libro de la estantería en la caja.
- si hay algún libro que no entra en la caja, dejar el libro en un montón aparte y probar con el siguiente.
- probar con todos los libros de la estantería, manteniendo el mismo orden de extracción de la estantería.
- los libros que no se han podido introducir en la caja, se vuelven a dejar en el mismo orden en la estantería.

```
public class BibliotecaEmbalada {
  ...
  public Caja embalar(Estanteria estanteria){
    MontonLibros monton = new MontonLibros();
    Caja caja = new Caja();
    Libro libro;
    while ((!estanteria.estaVacia()) && (caja.haySitio())){
      libro = estanteria.extraer();
      libro = caja.introducir(libro);
      if (libro != null)
        monton.añadir(libro);
    }
    while (!monton.estaVacio()){
      libro = monton.quitar();
      estanteria.introducir(libro);
    }
    return caja;
  }
}

public class Estantería {
   ...
  public boolean estaVacia(){
    return libros.isEmpty();
  }

  public Libro extraer(){
    Libro libro = null;
    if (!libros.isEmpty())
      libro = (Libro) libros.pop();
    return libro;
  }
  public void introducir(Libro libro){
    libros.push(libro);
  }
}
```

```
public class Caja {
   /** Devuelve true si existe algún montón que tenga una altura inferior
     * a ALTURA_MAX y false, en caso contrario, es decir, cuando todos los
     * montones tengan la altura igual a ALTURA_MAX
     */
   ...
   public boolean haySitio(){
     int ind = 0;
     while ((ind < NUM_MONTONES_MAX) &&
            (montones[ind].getAltura() == ALTURA_MAX))
       ind++;
     return ind < NUM_MONTONES_MAX;
   }

   /** La regla que se sigue para introducir un libro en una caja es la de
     * encontrar un montón que tenga el hueco suficiente para introducir
     * el libro dado, es decir, un montón que tenga un hueco libre mayor o
     * igual que el grosor del libro dado. Sin mover los libros que ya
     * están en los montones. Si el libro no entra se devuelve y si entra
     * se devuelve vacio, es decir, null.
     */
   public Libro introducir(Libro libro){
     int ind = buscarMonton(libro);
     if (ind != -1) {
       montones[ind].añadir(libro);
       return null;
     }
     return libro;
   }

   /** Siguiendo el criterio de introducir que se ha comentado arriba,
     * Devuelve un valor entre 0 y NUM_MONTONES_MAX-1 si hay sitio en
     * alguno de los montones para introducir el libro dado y -1, en caso
     * contrario.
     */
   private int buscarMonton(Libro libro){
     int ind = 0;
     while ((ind < NUM_MONTONES_MAX) &&
            (montones[ind].getAltura() + libro.getGrosor() > ALTURA_MAX))
       ind++;
     if (ind == NUM_MONTONES_MAX) // no hay sitio en la caja para el libro dado
       ind = -1;

     return ind;
  }
}

public class MontonLibros{
   ...
   public void añadir(Libro libro){
     altura += libro.getGrosor();
     numLibros ++;
     libros.push(libro);
   }
}
```