CONVOCATORIA EXTRAORDINARIA

ESTRUCTURA DE DATOS Y ALGORITMOS, CURSO 2012/2013

17 JUNIO 2013

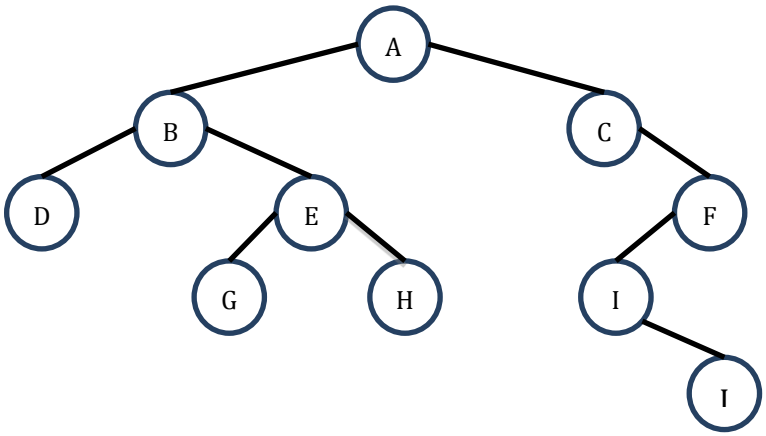
Apellidos: _____ **Nombre** _____

NIA: _____ **Grupo:** _____

Algunas reglas:

- Antes de comenzar el examen, escribe tu nombre y grupo.
- Lee atentamente el enunciado de cada ejercicio.
- Utiliza las hojas de examen (hojas de cuadros) para preparar tu respuesta. Una vez que estés segur@ de la solución de un ejercicio, escribe la respuesta en el hueco que le corresponde.

1. (**1 punto**) El recorrido de un árbol binario en doble orden se define como sigue:
Si el árbol binario está vacío, no se hace nada, de lo contrario:
a) visitar la raíz, por primera vez;
b) recorrer el subárbol izquierdo en doble orden;
c) visitar la raíz, por segunda vez;
d) recorrer el subárbol derecho en doble orden;
Por ejemplo, dado el siguiente árbol binario:



Su recorrido en doble orden sería: A1,B1,D1,D2,B2,E1,G1,G2,E2,H1,H2,A2,C1,C2,F1,I1,I2,J1,J2,F2
donde A1 significa que A está siendo visitado por primera vez, etc.

Se pide:

a) (0.75 ptos) Implementar un método denominado *showDoubleOrder()* en la clase BinTreeNode (implementación de un nodo de árbol binario), que imprima por pantalla el doble orden del subárbol que cuelga del nodo.

Solución:

```
public void showDoubleOrder() {
                showDoubleOrder(this);  }

static<E> void showDoubleOrder(BinTreeNode<E> node) {
        if (node != null){
                System.out.println(node.getElement());
                showDoubleOrder(node.leftChild);
                System.out.println(node.getElement());
                showDoubleOrder(node.rightChild);
        }
}
```

b) (0.25 ptos) Modifica el método anterior para que devuelva una lista que almacene el recorrido en doble orden del subárbol. Llama al nuevo método *getDoubleOrder()*

```
public IList<E> getDoubleOrder() {
        IList<E> list=new SList<E>();
         getDoubleOrder(this, list);
         return list;
}

static <E> void getDoubleOrder(BinTreeNode<E> node, IList<E> list) {
        if (node!= null)
                list.addLast(node.getElement());
                getDoubleOrder(node.leftChild,list);
                list.addLast(node.getElement());
                getDoubleOrder(node.rightChild,list);
        }
}
```

2. (0.5 punto) Escribe un método denominado *colaToPila* que reciba una cola y que devuelva una pila con los mismo elementos y en el mismo orden, es decir, el primer elemento de la cola ha de convertirse en la cima de la pila, y el último elemento de la cola debe estar en el fondo de la pila. Se cuenta con la clases SStack y SQueue que contiene las operaciones básicas definidas para pilas y colas respectivamente. No puede utilizar otras estructuras de datos auxiliares.

Solución:

```
public static SStack<Integer> colaToPila(SQueue<Integer> colaEnteros){

        SStack<Integer> pilaEnteros = new SStack<Integer>();

        while(!colaEnteros.isEmpty()) pilaEnteros.push(colaEnteros.dequeue());

        while(!pilaEnteros.isEmpty()) colaEnteros.enqueue(pilaEnteros.pop());

        while(!colaEnteros.isEmpty()) pilaEnteros.push(colaEnteros.dequeue());

        return pilaEnteros;
}
```

3. (0.5 punto) Escribe un método denominado *invertirCola* que dada una cola de números enteros, devuelva la cola invertida. Se cuenta con la clase SQueue que contiene las operaciones básicas definidas para colas. Puedes utilizar estructuras de datos auxiliares para hacerlo.

Solución:

```
public static SQueue<Integer> invertirCola(SQueue<Integer> colaEnteros){

       SStack<Integer> pilaAux = new SStack<Integer>();

       while(!colaEnteros.isEmpty())   pilaAux.push(colaEnteros.dequeue());

      while(!pilaAux.isEmpty())   colaEnteros.enqueue(pilaAux.pop());

      return colaEnteros;

  }
```

4. (0.5 punto) Escribe un método denominado *incrementarPila* que dada una pila de enteros y un número entero, incremente cada uno de los elementos de la pila con dicho valor y devuelva la pila actualizada. Se cuenta con la clase SStack que contiene las operaciones básicas definidas para pilas.

Solución:

```
public static IStack<Integer> incrementarPila(IStack<Integer> pilaEnteros, int k){

      SStack<Integer> pilaAux = new SStack<Integer>();

       //incrementamos cada elemento de la pila y lo insertamos en la pila auxiliar

       while(!pilaEnteros.isEmpty())    pilaAux.push(pilaEnteros.pop()+k);

       //volvemos a mover a la pila original para que conserve el mismo orden

      while(!pilaAux.isEmpty())     pilaEnteros.push(pilaAux.pop());

      return pilaEnteros;

  }
```

5 (1 punto) Desarrolla un método recursivo que elimine todos los elementos de una pila de enteros. El método debe imprimir por pantalla cada elemento que elimina. En la figura se muestra un ejemplo de ejecución para la pila {12,7,5,3}, donde el elemento 12 está en la cima.

```
Contenido inicial de la pila: [12,7,5,3]
Eliminando:[12]
Eliminando:[7]
Eliminando:[5]
Eliminando:[3]
Pila vacía
```

solución:

```
public static void vaciarPila(SStack<Integer> pila){
                    if((pila != null)&&(!pila.isEmpty())){
                              Integer eliminado = pila.pop();
                              System.out.println("Eliminando:[" + eliminado + "]");
                              vaciarPila(pila);
                    }
          }
```

¿El método que has definido podría ser NO estático? ¿Por qué?

Los métodos de objeto (no estáticos) son aquellos que acceden o modifican los atributos del objeto. Si un método no modifica ni consulta miembros del objeto, el método **debe ser estático**.
En este ejercicio, el método recibe la pila que debe ser vaciada y no consulta ni modifica ningún atributo del objeto, por tanto, el método debe ser estático.
Nota importante: Java permite crear métodos no estáticos aunque no modifiquen ni consulten elementos de objeto, pero es importante saber que esta práctica no es lo más correcto.

6. (1.5 puntos) Implementa un algoritmo recursivo que determine si una cadena de caracteres tiene mayor número de dígitos que de letras. La salida debe ser 0 si el número de dígitos es igual al de letras; 1 si el número de dígitos es mayor que el de letras y -1 en otro caso. En la figura se muestra un ejemplo de ejecución para la cadena *34567rtyu*.

Cadena de entrada: [34567rtyu]
La cadena: [34567rtyu] tiene MENOR número de letras que de dígitos → Devuelve 1

Recuerda que la clase *Character* tiene un método estático *isDigit* que tiene la siguiente definición:

```
boolean java.lang.Character.isDigit(char ch)
```

También conviene recordar que la clase *String* dispone de un método denominado *substring* para obtener parte de la cadena original definido como:

```
java.lang.String.substring(int beginIndex, int endIndex)
```

Pista: el método *substring* te ayudará simplificar el problema en cada llamada recursiva

Solución:

```
public static int cuentaDigitosLetras(String s, int numDigitos, int numLetras){
        if(s == null){
                if(numDigitos == numLetras)  return 0;
                if(numDigitos > numLetras)  return 1;
                else return -1;
        }else{
                char arrayCaracteres[] = s.toCharArray();

                if((arrayCaracteres == null)||(arrayCaracteres.length == 0)){
                        return cuentaDigitosLetras(null,numDigitos,numLetras);
                }
                if(Character.isDigit(arrayCaracteres[0])){
                        numDigitos++;
                }else{
                        numLetras++;
                }

                //Hay que eliminar el carácter utilizado para la siguiente llamada recursiva
                String sresto = s.substring(1,s.length());
                return cuentaDigitosLetras(sresto,numDigitos,numLetras);
        }
}
```

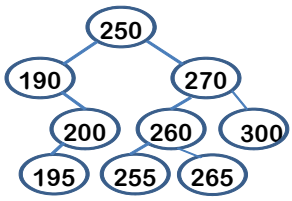
7 (1,5 puntos) Dado un nodo perteneciente a un árbol binario de búsqueda, desarrollar un método que devuelva el siguiente nodo en el recorrido preorder del árbol al que pertenece, o null en caso de que sea el último de dicho recorrido.

Pistas:

1) El método deseado no es recursivo y, aunque no figure en el enunciado, es estático.

2) En la casuística, es necesario considerar:

a) El nodo tiene hijo izquierdo. Por ejemplo, el siguiente a 250 es 190.



b) El nodo no tiene hijo izquierdo pero sí derecho. Por ejemplo, el siguiente a 190 es 200.

c) El nodo no tiene hijos ni izquierdo ni derecho. En este caso, será necesario ascender por el padre las veces que sea necesario, hasta encontrar un nodo que tenga hijos izquierdo y derecho, al que hayamos ascendido por su hijo izquierdo, y devolver su hijo derecho. Por ejemplo, para 195 debemos ascender al 250 y devolver 270, para 255 ascenderemos sólo al 260 y para devolver 265, y desde 265 ascenderemos a 270 y devolveremos 300.

d) En el caso c) puede ocurrir que no haya siguiente. Por ejemplo, 300 no tiene siguiente.

Solución:

```
static <K extends Comparable<K>, E> BSTNode<K, E> getPreorderNext(BSTNode<K, E> node) {
        // opcional
        if (node == null) return null;
        // caso a
        if (node.getLeftChild() != null) return node.getLeftChild();
        // caso b
        if (node.getRightChild() != null) return node.getRightChild();
        // caso c
        while (node.getParent() != null) {
          if (node.getParent().getLeftChild() == node &&
              node.getParent().getRightChild() != null) {
                    return node.getParent().getRightChild();
          }
          node = node.getParent();
        }
        // caso d - finalmente, si no hemos encontrado
        return null;
}
```

8. (1,5 puntos) Recordando la implementación del método *getAt(index)* de las listas, que devuelve el elemento alojado en la posición "index" de una lista dada. Se pide:

a) **(1 punto)** Implementar el mismo método para las listas doblemente enlazadas (clase DList), pero haciéndolo más eficiente que la implementación original, mostrada a continuación. El nuevo método lo llamaremos *getAtEfficient*. IMPORTANTE: para el nuevo método se podrá hacer uso del método *getSize()*, suponiendo que su complejidad es igual a 1. Además para el nuevo método vamos a suponer que index=1 implica devolver el primer elemento de la lista.

```
@Override
public E getAt(int index) {
        int i = 0;
        for (DNode<E> nodeIt = header.nextNode; nodeIt != trailer; nodeIt = nodeIt.nextNode) {
                if (i == index) {
                        return nodeIt.getElement();
                }
                ++i;
        }
        System.out.println("DList: Get out of bounds");
        return null;
}
```

Solución:

```
public E getAtEfficient(int index) {
        int size = this.getSize();
        int i = 1;
        if ((index > size) || (index <= 0)) { // ERROR: fuera de rango
            System.out.println("DList: Get out of bounds");
            return null;
        } else if (index <= size / 2) { // Recorre la lista en la primera mitad desde la cabecera
            for (DNode<E> nodeIt = header.nextNode; nodeIt != trailer; nodeIt = nodeIt.nextNode) {
                if (i == index) {
                        return nodeIt.getElement();
                }
                ++i;
            }
        } else { // Recorre la lista en la segunda mitad desde la cola
            for (DNode<E> nodeIt = trailer.previousNode; nodeIt != header; nodeIt = nodeIt.previousNode) {
                System.out.println(nodeIt.getElement());
                if (i == size - index + 1) {
                        return nodeIt.getElement();
                }
                ++i;
            }
        }
        return null;
}
```

b) **(0,25 puntos) ¿**Cuál es la complejidad del método *getAt*?

Solución: Peor caso: el elemento a devolver es el último de la lista → O(n)

c) **(0,25 puntos) ¿**Cuál es la complejidad del método *getAtEfficient*?

Solución: Peor caso: el elemento a devolver está en la mitad de la lista → O(n/2)

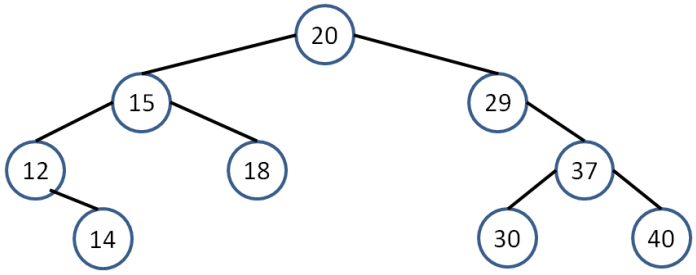
9. (1 punto) Sea un árbol binario de búsqueda que almacena los pares <edad, nombre> de los estudiantes de Informática en la Universidad Carlos III. Se pide implementar un método **recursivo** que devuelva en una lista los nombres de todos los estudiantes cuya edad esté comprendida en un rango de valores. Importante: el método ha de ser recursivo y la lista ha de devolverse ordenada según edad **de menor a mayor**. Este algoritmo estará formado por dos métodos, como se describen.

Solución:

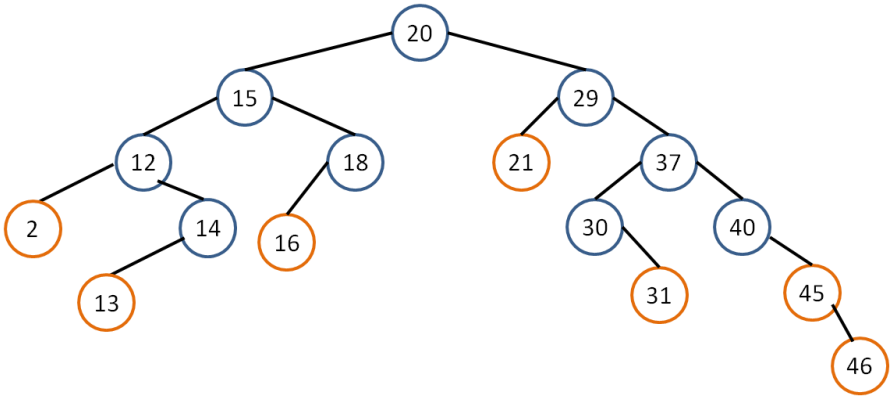
```
public class ArbolEdadEstudiantes extends BSTree<Integer, String> {

    /**
     * Método que devuelve la lista de nombres de estudiantes con una
     * edad comprendida entre el rango de valores introducido como parámetro
     */
    public IList<String> devuelveNombresPorEdad(int edadMin, int edadMax) {
       SList<String> lista = new SList<String>();
       devuelveNombresPorEdad(root, edadMin, edadMax, lista);
       return lista;
    }

    /**
     * Método recursivo que devuelve la lista de nombres de estudiantes con una
     * edad comprendida entre el rango de valores introducido como parámetro
     */
     public static void devuelveNombresPorEdad(
               BSTNode<Integer, String> nodo,
               int edadMin, int edadMax,
               IList<String> listaNombres) {

        /* es necesario recorrer el árbol en IN-ORDER, pero de forma simétrica (primero
            Árbol derecho, luego raíz, finalmente árbol izquierdo), para ordenar de menor
           a mayor, insertando al comienzo de la lista, ya que este método es
           eficiente que insertar al final de la lista */

        if (nodo != null) { // si el nodo es nulo acaba la recursión

           // si la key es menor o igual que la edad que busco --> busco en árbol dcho
           if (nodo.getKey() <= edadMax)
                devuelveNombresPorEdad(nodo.getRightChild(),
                                       edadMin, edadMax, listaNombres);
```

```
            // si el nodo está en el rango, inserto en lista
            if ((nodo.getKey() >= edadMin)&&(nodo.getKey() <= edadMax))
                              listaNombres.addFirst(nodo.getElement());

            // si la key es mayor o igual que la edad que busco --> busco en árbol izdo
            if (nodo.getKey() >= edadMin)
                 devuelveNombresPorEdad(nodo.getLeftChild(),
                                        edadMin, edadMax,listaNombres);
        }
     }
}
```

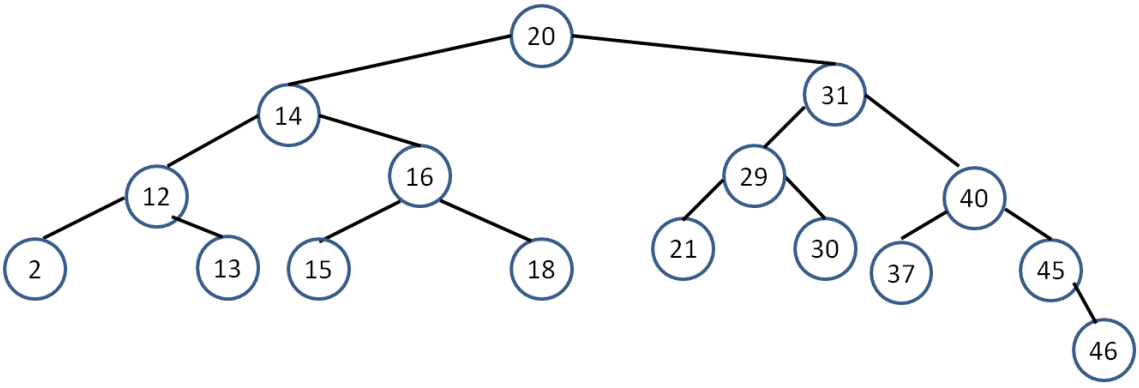
10 (0.5 puntos) Dado el siguiente árbol Binario de Búsqueda donde la clave es igual al propio elemento almacenado, se pide:



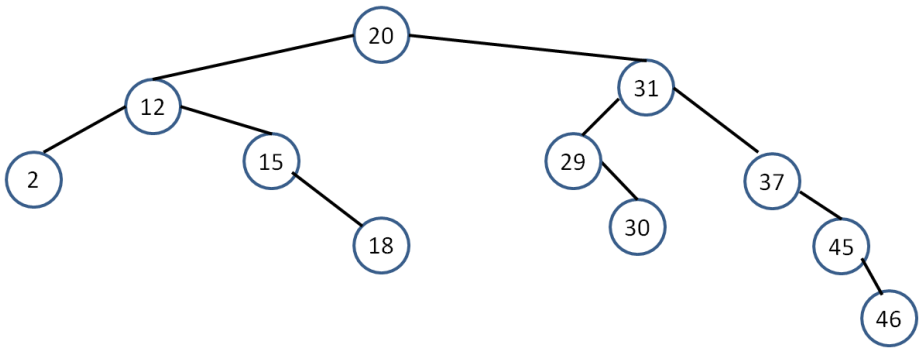
a) Realice las siguientes inserciones: 21, 13, 16, 31, 45, 46 y 2. Si estos elementos se insertaran en otro orden, ¿el árbol resultante sería diferente?



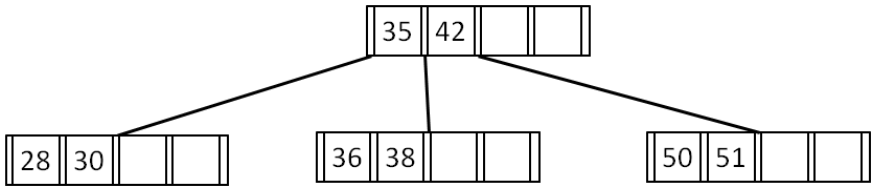
b) Realice el equilibrado en tamaño del árbol resultante del apartado 10a.



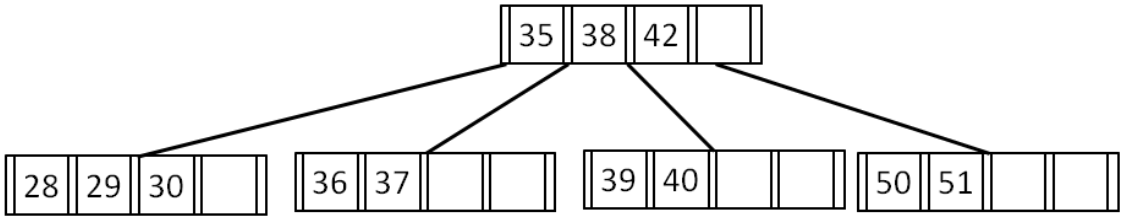
c) Utilizando como base el árbol resultante en el apartado anterior, elimine los nodos 16, 21, 40, 13, 14. NOTA: cada vez que se elimina un nodo, vamos a utilizar el subárbol izquierdo para rellenar (estrategia del predecesor).



11 (0.5 puntos) Dado el siguiente árbol B de orden 4 donde la clave es igual al propio elemento almacenado, se pide:

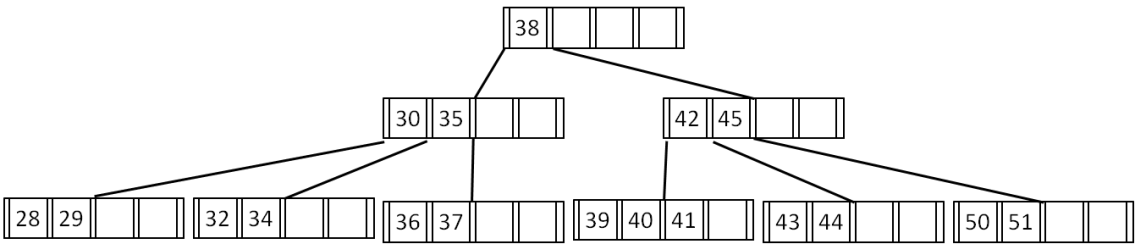


a) Realice las siguientes inserciones: 29, 40, 39 y 37. **Si estos elementos se insertaran en otro orden, ¿el árbol resultante sería diferente?**



El resultado sería el mismo si se cambia el orden

b) En el árbol resultante del apartado anterior (11a), realizar las siguientes inserciones: 43, 44, 41, 45, 34 y 32



c) Utilizando como base el árbol resultante en el apartado anterior (11b), realice las siguientes eliminaciones: 35, 30, 32, 38 y 42, teniendo en cuenta que se tomará la estrategia del predecesor en caso de ser necesario sustitución.