EXAMEN FINAL ESTRUCTURA DE DATOS Y ALGORITMOS

20 MAYO 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. (**1.5 puntos**) Implementa un método en la clase BinTreeNode (implementación de un nodo de árbol binario), que devuelva en una lista el recorrido por niveles del subárbol que cuelga del nodo.

Solución:

```
class BinTreeNode<E> {
        public IList<E> getLevelOrder() {
                IList<E> list = new SList<E>();
                getLevelorder(this, list);
                return list;
        }

        /**
         * Método iterativo para recorrer el árbol en orden por niveles
         * @param node
         * @param list
         */
        static<E> void getLevelOrder(BinTreeNode<E> node, IList<E> list) {

                if (node==null) return; //Árbol vacío

                //Creo una cola que nos va a permitir ir almacenando los nodos del árbol
                SQueue<BinTreeNode<E>> colaAux = new SQueue<BinTreeNode<E>>();
                colaAux.enqueue(node); //encolamos el primero nodo del subárbol

                while(!colaAux.isEmpty()){
                    //sacamos el primer elemento de la cola y lo añadimos a la lista
                        BinTreeNode<E> bn = colaAux.dequeue();
                        list.addLast(bn.getElement());

                        //ahora tenemos que encolar sus hijos para trararlos en la siguiente iteración
                        if(bn.leftChild!=null)  colaAux.enqueue(bn.leftChild);
                        if(bn.rightChild!=null)         colaAux.enqueue(bn.rightChild);

                }
        }
}
```

2. (**1,5 puntos**) Implementa un algoritmo que reciba una pila de enteros, en la cual los elementos de la pila de entrada están siempre ordenados de manera que el mayor está en la cima de la pila, y obtenga los elementos pares de la pila. Dichos elementos deben ser devueltos en una cola, de manera que el mayor quede en la cabeza de la cola (front) y el menor en la posición final de la misma (tail).
Ejemplo – Pila de entrada: 1,2,3,4,5,6,7,8,9,10 (top) Cola de salida: (front) 10,8,6,4,2 (tail)

a) Implementa el algoritmo de tal forma que, al acabar, la pila de entrada quede vacía. (0.75 puntos)

Solución:

```
IQueue<Integer> eliminarPares(IStack<Integer> pilaEnteros){
        SQueue<Integer> colaPares = new SQueue<Integer>();

        while(!pilaEnteros.isEmpty()){
                Integer entero = pilaEnteros.pop();
                if(entero.intValue()%2 == 0){
                        colaPares.enqueue(entero);
                }
        }
        return colaPares;
}
```

b) Modifica el algoritmo para que en la pila de entrada no quede vacía; en ella quedarán almacenados los elementos impares, **en el mismo orden** en el que estaban. (0.75 puntos)

Solución:

```
    public static IQueue<Integer> eliminarPares(IStack<Integer> pilaEnteros){

      SQueue<Integer> colaPares = new SQueue<Integer>();
      SStack<Integer> pilaImpares = new SStack<Integer>();

      while(!pilaEnteros.isEmpty()){
         Integer entero = pilaEnteros.pop();

         if(entero.intValue()%2 == 0){
             colaPares.enqueue(entero);
         } else {
              pilaImpares.push(entero);
         }
      }

     while(!pilaImpares.isEmpty()){
       pilaEnteros.push(pilaImpares.pop());
     }
     return colaPares;
  }
}
```

3. **(2 puntos)** (máximo **1 punto**, si la solución se hace iterativa):

a) Implementa un **algoritmo recursivo** que reciba como parámetro un nodo de un árbol binario de búsqueda y muestre por pantalla las claves de los nodos que hay en el camino **desde la raíz** del árbol hasta el nodo (ambos nodos deben estar incluidos). (**0,8 puntos** recursivo**, 0,4 puntos** iterativo)

Solución:

```
public static<K extends Comparable<K>, E> void mostrarCamino (BSTNode<K,E> node) {
        if (node!=null) {
           mostrarCamino(node.getParent());
           System.out.println(node.getKey());
        }
}
```

b) Modifica el algoritmo para que, en lugar de mostrar los elementos por pantalla, los almacene en una lista, y la devuelva. (**0,6 puntos** recursivo**, 0,4 puntos** iterativo)
Nota: Se recomienda usar un método intermedio para la solución recursiva.

Solución:

```
static<K extends Comparable<K>, E> IList<K> guardarCaminoLista (BSTNode<K,E> node) {
                DList<K> lista=new DList<K>();
              guardarCaminoLista(node,lista);
              return lista;
        }


      static<K extends Comparable<K>, E> void guardarCaminoLista (BSTNode<K,E> node, DList<K> lista) {
              if (node!=null) {
                 guardarCaminoLista(node.getParent());
                 lista.addLast(node.getKey());
              }
      }
```

c) Modifica el primer algoritmo para que además de mostrar la clave de cada nodo muestre su profundidad. Se prohíbe utilizar el método getDepth. (**0.6 puntos, 0,2 puntos** iterativa)
Sugerencia para la solución recursiva: El método debería devolver la profundidad del nodo.

Solución:

Solución:

```
 static<K extends Comparable<K>, E> int mostrarCaminoDepth (BSTNode<K,E> node) {
              if (node==null) {
                        return 0;
              }
               int depth = mostrarCaminoDepth(node.getParent());
               System.out.println(depth+": "+node.getKey());
               return depth+1;
  }
```

4. (**1,5 puntos**) Implementa un método iterativo en la clase BSTNode (implementación de un nodo de árbol binario) que reciba como argumento una clave y un nodo. El método debe buscar dicha clave en el subárbol que cuelga del nodo y devolver el elemento asociado a esa clave. Si no existe ningún nodo con dicha clave el método deberá devolver null. ¿Cuál es la complejidad del método?

Solución:

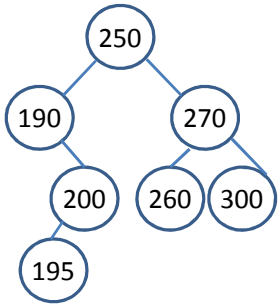
```
static<K extends Comparable<K>,E> E searchIte(BSTNode<K, E> node, K key) {
              BSTNode<K, E> nodeIt=node;
              while (nodeIt!=null) {

                        K keyNode=node.getKey();
                        if (keyNode.equals(key)) return nodeIt.getElement();

                        if (key.compareTo(keyNode)<0) nodeIt=nodeIt.leftChild;
                        else nodeIt=nodeIt.rightChild;

              }

              return null;
      }
```

Complejidad: O(n), siendo n la altura del árbol

5. (**1,5 puntos**) Sea un árbol binario que almacena los pares <precio, ubicación> de habitaciones en pisos de alquiler cerca de la Universidad, por orden de precio. Se pide:

a) (**0,2 puntos**.) Dibuja el árbol resultante de la inserción de los siguientes pares en secuencia:
<250, “Avda. Universidad, 1”>,
<190, “c/Batalla Lepanto, 1”>,
<200, “c/Río Manzanares, 1”>,
<195,”c/Butarque, 1”>,
<270,”c/ Madrid, 1”>,
<300, “c/ San Nicasio, 1”> y
<260, “c/Río Guadiana, 1”>
Nota: Basta con indicar el precio en el dibujo.

Solución:



b) (**0,3 puntos**.) Crea una clase para almacenar cada habitación en un árbol binario de búsqueda. Será necesario para ello hacer uso de las librerías sobre Árboles Binario de Búsqueda proporcionadas durante el curso (en particular, de la BSTree).

Solución:

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

}
```

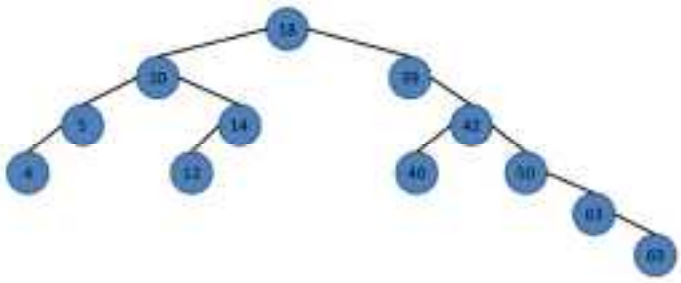
c) **(1 punto)** Implementa un algoritmo **recursivo** que reciba un nodo y un precio y devuelva en una lista las ubicaciones de las habitaciones en alquiler menores que dicho precio. Importante: el método ha de ser recursivo y la lista ha de devolverse ordenada según precio **de mayor a menor**. Este algoritmo estará formado por dos métodos, como se describe.

Solución:

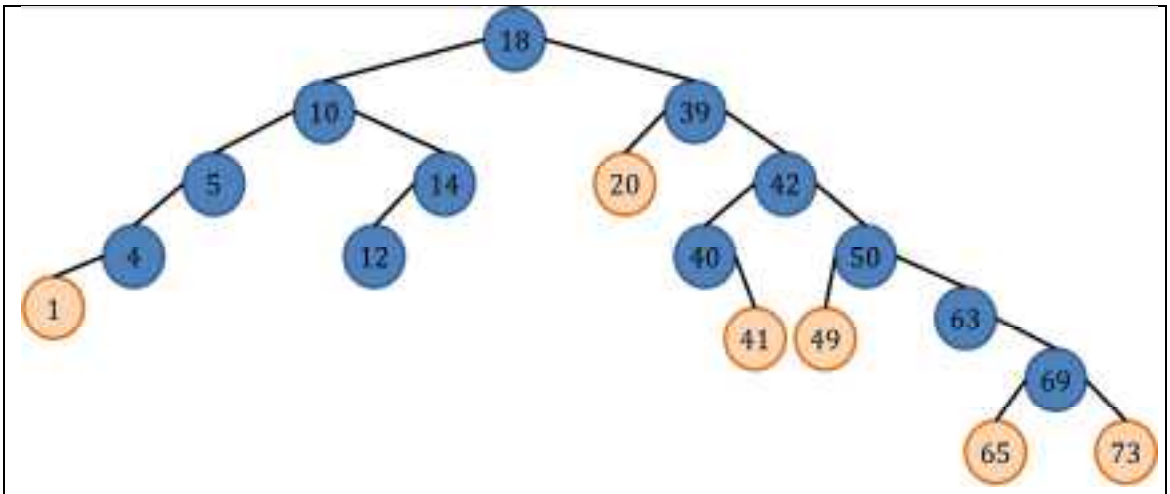
```
public class ArbolHabitaciones extends BSTree<Integer, String> {

    /**
     * En ArbolHabitaciones: Método que devuelve la lista de habitaciones con un
     * precio por debajo de PrecioMax pasado como argumento
     */
    public IList<String> devuelveUbicacionesPorPrecio(int PrecioMax) {
        SList<String> lista = new SList<String>();
        devuelveUbicacionesPorPrecio(root, PrecioMax, lista);
        return lista;
    }

    /**
     * Método que devuelve la lista de habitaciones con un precio por debajo de
     * PrecioMax pasado como argumento
     */
    public static void devuelveUbicacionesPorPrecio(
            BSTNode<Integer, String> nodo, int PrecioMax,
            IList<String> listaUbicaciones) {
        // si es nulo acaba la recursión
        if (nodo != null) {

            // es necesario recorrer el árbol en PRE-ORDER para ordenar de mayor
            // a menor, insertando al comienzo de la lista
            devuelveUbicacionesPorPrecio(nodo.getLeftChild(), PrecioMax,
                    listaUbicaciones);
            if (nodo.getKey() <= PrecioMax) { // inserto en lista
                listaUbicaciones.addFirst(nodo.getElement()); // Anyade al
                                                              // principio
            }
            // si la key es menor que el precio que necesito, busco también en
            // árbol dcho
            if (nodo.getKey() < PrecioMax) {
                devuelveUbicacionesPorPrecio(nodo.getRightChild(), PrecioMax,
                        listaUbicaciones);
            }
        }
    }
```

NOTA: El segundo comentario del método contiene un error. No es recorrido pre-order, sino in-order.

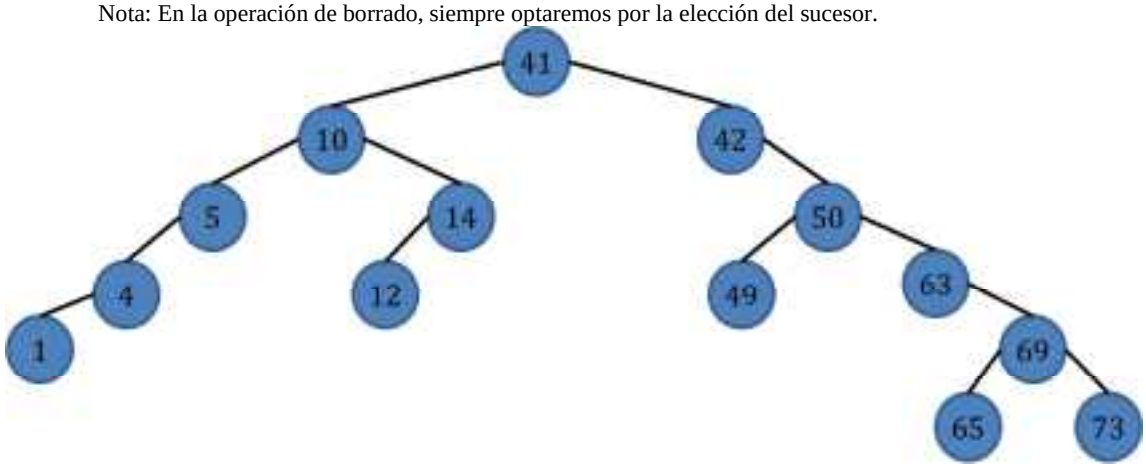
6. (**1 punto**) Dado el siguiente árbol binario de búsqueda donde la clave es igual al propio elemento almacenado, se pide:



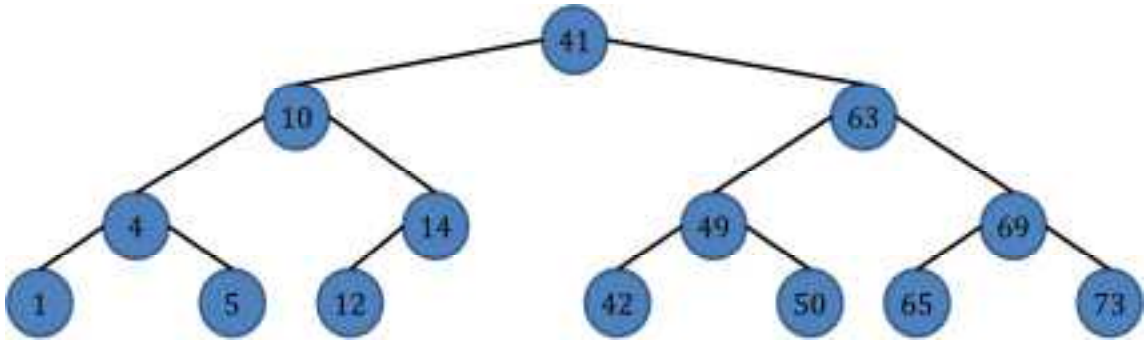
a) Realiza las siguientes inserciones: 49, 73, 20, 1, 65 y 41. Si estos elementos se insertaran en otro orden, ¿el árbol resultante sería diferente? Dibuje a continuación únicamente el árbol resultante (las operaciones previas puedes escribirlas en las hojas de cuadro) (**0,2 puntos**)



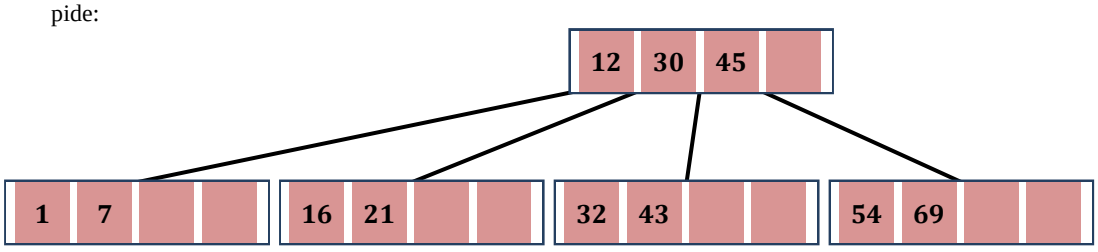
b) Utilizando como base el árbol resultante en el apartado anterior, elimina **cuatro veces** la raíz del árbol. Dibuja a continuación únicamente el árbol resultante (las operaciones previas puedes escribirlas en las hojas de cuadro). (**0,4 puntos**)
Nota: En la operación de borrado, siempre optaremos por la elección del sucesor.



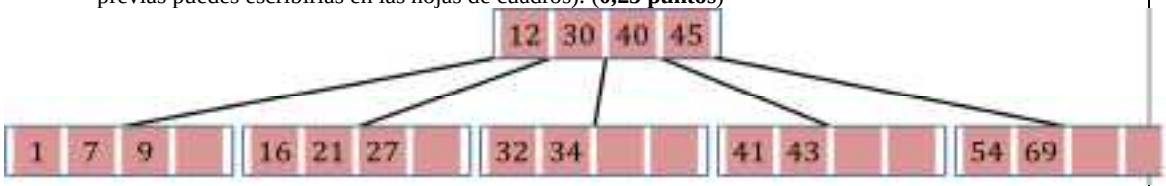
c) Realiza el equilibrado en tamaño del árbol resultante del apartado anterior. Dibuja a continuación únicamente el árbol resultante (las operaciones previas puedes escribirlas en las hojas de cuadro). (**0,4 puntos**)
Nota: En la operación de borrado, siempre optaremos por la elección del sucesor.



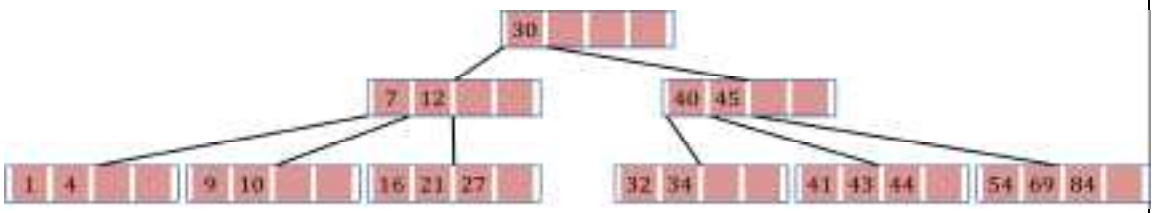
7. (**1 punto**) Dado el siguiente árbol B de orden 4, donde la clave es igual al propio elemento almacenado, se pide:



a) Realiza las siguientes inserciones: 34, 40, 9, 41 y 27. Si estos elementos se insertaran en otro orden, ¿el árbol resultante sería diferente? Dibuja a continuación únicamente el árbol resultante (las operaciones previas puedes escribirlas en las hojas de cuadros). (**0,25 puntos**)



b) En el árbol resultante del apartado a), realiza las siguientes inserciones: 84, 44, 4 y 10. Dibuja a continuación únicamente el árbol resultante (las operaciones previas puede escribirlas en las hojas de cuadro). (**0.25 puntos**)



c) En el árbol resultante del apartado b), realiza las siguientes eliminaciones: 44, 45, 10 y 4. Dibuja a continuación únicamente el árbol resultante (las operaciones previas puedes escribirlas en las hojas de cuadro). (**0,5 puntos**)