Exámen 2012, preguntas y respuestas - Resolucion de la Recuperacion del Segundo Parcial

Estructuras de datos y algoritmos (Universitat Politecnica de Valencia)

1.- Las siguientes clases implementan un ABB en Java:

```
public class ABB <E extends Comparable <E>> {
   protected NodoABB<E> raiz;
   protected int talla;
   public ABB(){ raiz = null; talla = 0; }
   //Resto de métodos de la clase
   …
}
class NodoABB<E> {
   E dato;
   NodoABB<E> izq, der;
   public NodoABB(E e){ dato = e; izq = null; der = null; }
   public NodoABB(E e, NodoABB<E> izq, NodoABB<E> der){
       dato = e; this.izq = izq; this.der = der;
   }
}
```

Se pide: **(4 puntos)**

(a) Diseñar un método en la clase *ABB* que devuelva el tamaño de aquel de los nodos de un ABB que contenga a *e* como dato, *0* si *e* no está en el ABB; suponer que no hay elementos repetidos en el ABB. **(3 puntos)**

Nota: el tamaño de un nodo se define como el número de descendientes que tiene, él mismo incluido, y el de un árbol como el de su nodo raíz.

```
public int tamañoNodo(E e){
    return tamanyo(recuperar(e, this.raiz));
}
protected NodoABB<E> recuperar(E e, NodoABB<E> actual){
    NodoABB<E> res = actual;
    if ( actual!=null ){
        int resC = actual.dato.compareTo(e);
        if       ( resC<0 )  res = recuperar(e, actual.der);
        else  if ( resC>0 )  res = recuperar(e, actual.izq);
    }
    return res;
}
protected int tamanyo(NodoABB<E> actual){
    if ( actual==null ) return 0;
    else return 1 + tamanyo(actual.izq) + tamanyo(actual.der);
}
```

(b) Indicar el coste Temporal del método diseñado, justificándolo adecuadamente. **(1 punto)**

Suponiendo que x es el tamaño del ABB, x=*talla*:

En el Mejor de los Casos *e* no está en el ABB y, además, es mayor (o menor) que todos los elementos del ABB Completamente Degenerado por la Izquierda (o Derecha) sobre el que se aplica el método *tamañoNodo*, pues entonces su coste se reduce al de *recuperar* un elemento de un nodo de *tamanyo* igual a *cero* (el hijo Izquierdo o Derecho del nodo raíz del ABB). Por tanto, $\mathbf{T^M_{tamaño}(x) \in \Theta(1)}$ **y** $\mathbf{T_{tamaño}(x) \in \Omega(1)}$.

En el Peor de los Casos *e* es el dato que ocupa la raíz del ABB sobre el que se aplica el método *tamañoNodo*, pues entonces su coste es el de *recuperar* un elemento del nodo raíz de un ABB de *tamanyo* igual a *talla*, independiente de su grado de Equilibrio. Por tanto, $\mathbf{T^P_{tamaño}(x) \in \Theta(x)}$ **y** $\mathbf{T_{tamaño}(x) \in O(x)}$.

2.- Supóngase que se ha modificado la clase genérica *TablaHash* para permitir la inserción de entradas con la misma clave. **Se pide** diseñar en esa clase un método *recuperarIguales* que, con coste mínimo, obtenga una Lista con Punto de Interés con los valores de todas las entradas de una Tabla Hash cuya clave sea *c*. **(3 puntos)**

```
public ListaConPI<V> recuperarIguales(C c){
    ListaConPI<V> res = new LEGListaConPI<V>();
    // todas las Entradas de la Tabla con clave c sólo pueden
    // estar en una cubeta: elArray[posTabla(c)]
    ListaConPI<EntradaHash<C,V>> lpi = elArray[posTabla(c)];
    for ( lpi.inicio(); !lpi.esFin(); lpi.siguiente() ){
          EntradaHash<C,V> e =  lpi.recuperar();
          if ( e.clave.equals(c) ) res.insertar(e.valor);
    }
    return res;
}
```

```
public interface
  ListaConPI<E> {
  void insertar(E e);
  void eliminar();
  E recuperar();
  void inicio();
  void siguiente();
  boolean esFin();
  boolean esVacia();
  void fin();
  int talla();
}
```

3.-Supóngase que los *talla*>0 elementos que contiene el atributo *elArray* de la clase *MonticuloBinari*o se han introducido usando el siguiente método, y no usando el método *insertar*:

```
public void introducir(E e){
   if ( talla==elArray.length-1 ) duplicarArray();
   elArray[++talla] = e;
}
```

Nótese entonces que, al menos, hay que comprobar si los elementos de *elArray* cumplen la Propiedad de Ordenación de un Min-Heap; si no lo hacen, como ya es sabido, será necesario invocar la ejecución del método *arreglar*. Para ello, s**e pide**: **(3 puntos)**

(a) Diseñar en la clase *MonticuloBinario* un método eficiente *incumplePO* tal que devuelva la posición del primer elemento de *elArray*, entre *1* y *talla*, que **in**cumpla la propiedad de ordenación del Min-Heap, *elArray.length* si no existe tal elemento. **(2 puntos)**

```
public int incumplePO(){
    for ( int i=2; i<=talla; i++ )
      if ( elArray[i].compareTo(elArray[i/2])<0 ) return i;
    return elArray.length;
}
```

(b) Indicar el coste Temporal del método diseñado, justificándolo adecuadamente. **(1 punto)**

Suponiendo que x es el tamaño del Min-Heap, x=*talla*:

En el Mejor de los Casos el Hijo Izquierdo de la raíz del Min-Heap, que ocupa la posición *2* de *elArray*, ya incumple la propiedad de ordenación (elArray[2]<elArray[1]). Así, $\mathbf{T^M_{incumplePO}(x) \in \Theta(1)}$ **y** $\mathbf{T_{incumplePO}(x) \in \Omega(1)}$.

En el Peor de los Casos, tras *talla* comparaciones, ninguno de los elementos introducidos en *elArray* incumple la propiedad de ordenación del Min-Heap. Así, $\mathbf{T^P_{incumplePO}(x) \in \Theta(x)}$ **y** $\mathbf{T_{incumplePO}(x) \in O(x)}$.