Ejercicio de seminario - soluciones de algunos ejercicios

Estructuras de datos y algoritmos (Universitat Politecnica de Valencia)

Ejercicio 1.- El siguiente método público **enQueNivel** de la clase **ABB** obtiene el nivel de un Árbol Binario de Búsqueda (ABB) en el que se encuentra cierto dato **e**, -1 si no está.

```
public int enQueNivel(E e){
  int res = -1;
  if ( this.raiz!=null ) res = enQueNivel(e, this.raiz, 0);
  return res;
}
// SII actual!=null
protected int enQueNivel(E e, NodoABB<E> actual, int nivelActual){
    int res = -1; //Caso base implícito: actual tiene un nodo Y el dato que contiene NO es e → devuelve -1
    if ( actual.dato.equals(e) )     res = nivelActual; // si encontrado e: actualizar resultado
    else{  // buscar e en el hijo Izquierdo de actual, si existe → buscar e en el nível siguiente al de actual
           if ( actual.izq!=null )    res = enQueNivel(e, actual.izq, nivelActual+1);
           // Si e no está en el hijo Izquierdo de actual →
           // buscar e en el hijo Izquierdo de actual, si existe → buscar e en el nível siguiente al de actual
           if ( res==-1 && actual.der!=null  )      res = enQueNivel(e, actual.der, nivelActual+1);
    }
    return res;
}
```

Suponiendo que el ABB está *equilibrado*, **se pide**:

(a) Indicar el tamaño del problema **x** que resuelve el método recursivo (protected).

x = tamaño del Nodo actual, el del ABB SOLO en la llamada más alta.

(b) Indicar si existen instancias significativas para dicho método recursivo y, en caso afirmativo, describir cada una de ellas.

Es un problema de **Búsqueda** y por lo tanto, **sí** hay instancias significativas. En concreto, **en el caso general**

Caso Mejor: el dato e es igual al que ocupa el Nodo actual, por lo que no hay que realizar ninguna llamada recursiva para devolver 0, el nivel del ABB en el que se encuentra e

Caso Peor: el dato e no está en el Nodo actual, por lo que hay que recorrer todos sus Caminos para constatarlo antes de devolver -1

(c) En función de la respuesta dada en el apartado anterior, escribir las Relaciones de Recurrencia que expresan la Complejidad Temporal de dicho método recursivo.

$T^M_{enQueNivel}(x>=1) = k_2$

$T^P_{enQueNivel}(x>1) = 2 * T^P_{enQueNivel}(x/2) + K_3;$ $T^P_{enQueNivel}(x=1) = k_1;$

(e) En función de las respuestas anteriores, y aplicando los teoremas de coste, indicar el coste Temporal Asintótico del método.

$T^M_{enQueNivel}(x) \in \Theta(1) \rightarrow T_{enQueNivel}(x) \in \Omega(1)$

$T^P_{enQueNivel}(x) \in \Theta(x)$ **por Teorema 3, con a=c= 2 y sobrecarga cte** → $T_{enQueNivel}(x) \in O(x)$

Teorema 1: $f(x) = a \cdot f(x - c) + b$, con $b \geq 1$
- si a=1, $f(x) \in \Theta(x)$;
- si a>1, $f(x) \in \Theta(a^{x/c})$

Teorema 2: $f(x) = a \cdot f(x - c) + b \cdot x + d$, con b y d≥1
- si a=1, $f(x) \in \Theta(x^2)$;
- si a>1, $f(x) \in \Theta(a^{x/c})$

Teorema 3: $f(x) = a \cdot f(x/c) + b$, con $b \geq 1$
- si a=1, $f(x) \in \Theta(\log_c x)$;
- si a>1, $f(x) \in \Theta(x^{\log_c a})$

Teorema 4: $f(x) = a \cdot f(x/c) + b \cdot x + d$, con b y d≥1
- si a<c, $f(x) \in \Theta(x)$;

```
protected int enQueNivel(E e, NodoABB<E> actual, int nivelActual){
    int res = -1;
    if ( actual.dato.equals(e) )     res = nivelActual;
    else{
        if ( actual.izq!=null )    res = enQueNivel(e, actual.izq, nivelActual+1);
        if ( res==-1 && actual.der!=null  )      res = enQueNivel(e, actual.der, nivelActual+1);
    }
    return res;
}
```

(f) Modificar el método recursivo analizado para hacerlo lo más eficiente posible; indicar también cuál es la mejora de coste conseguida y por qué se logra.

```
protected int enQueNivel( E e,  NodoABB actual, int nivelActual ){
    int res = -1;
    resC = actual.dato.compareTo(e);
    if ( resC == 0 )                                res = nivelActual;
    else
        if ( resC>0 && actual.izq!=null )           res = enQueNivel(e, actual.izq, ++nivelActual);
        else if ( actual.der!=null )                res = enQueNivel(e, actual.der, ++nivelActual);
    return res;
}
```

Aunque el Análisis por Casos coincide, en el Peor de los Casos se busca e en un Camino del ABB en lugar de en todos. Ello supone realizar una llamada en el caso general en lugar de dos y, por tanto, que en este caso el coste del método sea logarítmico en lugar de lineal.

Ejercicio 8.- Analiza el coste del siguiente método, que devuelve el nº de elementos de un Nodo de un ABB que son mayores que uno dado e.

```
protected int contarMayoresQue(E e, NodoABB<E> actual){
        int res =0;
        if ( actual!=null ){
                if ( actual.dato.compareTo(e)>0 ) res += 1;
                res += contarMayoresQue(e, actual.izq)
                res += contarMayoresQue(e, actual.der);
        }
        return res;
}
```

En concreto**,** suponiendo que el ABB está *equilibrado*, **se pide**:

(a) Indicar el tamaño del problema **x** que resuelve el método.

x = tamaño del Nodo actual.

(b) Indicar si existen instancias significativas para dicho método y, en caso afirmativo, describir cada una de ellas.

Es un problema de **Recorrido** y por lo tanto, **NO** hay instancias significativas.

(c) En función de la respuesta dada en el apartado anterior, escribir las Relaciones de Recurrencia que expresan la Complejidad Temporal de dicho método recursivo.

$T_{contarMayoresQue}(x=0) = k_1$

$T_{contarMayoresQue} (x>0) = 2 * T_{contarMayoresQue} (x/2) + K_1;$

(e) En función de las respuestas anteriores, y aplicando los teoremas de coste, indicar el coste Temporal Asintótico de dicho método recursivo.

$T_{contarMayoresQue} \in \Theta(x)$ **por Teorema 3, con a=c= 2 y sobrecarga cte**

Teorema 1: $f(x) = a \cdot f(x - c) + b$, con $b \geq 1$
- si a=1, $f(x) \in \Theta(x)$;
- si a>1, $f(x) \in \Theta(a^{x/c})$

Teorema 2: $f(x) = a \cdot f(x - c) + b \cdot x + d$, con b y d≥1
- si a=1, $f(x) \in \Theta(x^2)$;
- si a>1, $f(x) \in \Theta(a^{x/c})$

Teorema 3: $f(x) = a \cdot f(x/c) + b$, con $b \geq 1$
- si a=1, $f(x) \in \Theta(\log_c x)$;
- si a>1, $f(x) \in \Theta(x^{\log_c a})$

Teorema 4: $f(x) = a \cdot f(x/c) + b \cdot x + d$, con b y d≥1
- si a<c, $f(x) \in \Theta(x)$;
- si a=c, $f(x) \in \Theta(x \cdot \log_c x)$;
- si a>c, $f(x) \in \Theta(x^{\log_c a})$

(g) Utilizar las propiedades del ABB, asumiendo que no existen elementos duplicados, para diseñar un método en la clase *ABB* para resolver el mismo problema pero, si es posible, de manera más eficiente. Luego, realiza el análisis de su coste y discute en base a él si el (nuevo) método diseñado mejoraría su eficiencia si se dispusiera de un ABB con Rango (cuyos nodos contienen el atributo talla que representa su tamaño), en lugar de un simple ABB.

```
protected int contarMayoresQue(E e, NodoABB<E> actual){
        int res =0;
        if ( actual!=null ){
                if ( actual.dato.compareTo(e)>0 ) res += 1;
                res += contarMayoresQue(e, actual.izq)
                res += contarMayoresQue(e, actual.der);
        }
        return res;
}
```

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

Alternativamente, se podría haber descrito como sigue, sin usar **tamanyo**(actual)

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

Análisis del coste del (nuevo) método que usa tamanyo(actual)**:**

i. Indicar el tamaño del problema **x** que resuelve el método.

x = tamaño del Nodo actual.

ii. Indicar si existen instancias significativas para dicho método recursivo y, en caso afirmativo, describir cada una de ellas

Al introducir las modificaciones, el un problema pasa a ser de **Búsqueda** y por lo tanto, **SÍ** hay instancias significativas. En concreto, **en el caso general ...**

Caso Mejor: el dato e es mayor que el máximo del Nodo actual, por lo que solo hay que recorrer la rama más a la derecha de actual, de longitud igual a la altura de actual e igual al logaritmo de su talla.

Caso Peor: el dato e es menor que todos los datos del Nodo actual, por lo que hay que contar todos sus nodos –contar el dato de actual (1) + calcular el tamaño de su hijo derecho (visitar del orden de x/2 nodos) y contar todos los de su hijo i izquierdo (visitar los restantes x/2 nodos de actual, aprox.)

iii. En función de la respuesta dada en el apartado anterior, escribir las Relaciones de Recurrencia que expresan la Complejidad Temporal del método.

$T_{contarMayoresQue}(x=0) = k_1$, **tanto en el Peor como en el Mejor de los casos;**

$T^M_{contarMayoresQue}(x>0) = 1 * T^M_{contarMayoresQue}(x/2) + k_2$;

$T^P_{contarMayoresQue}(x>0) = 1 * T^P_{contarMayoresQue}(x/2) + T_{tamanyo}(x/2)$

$= 1 * T^P_{contarMayoresQue}(x/2) + k_3 * x/2 = 1 * T^P_{contarMayoresQue}(x/2) + k_4 * x$;

iv. En función de las respuestas anteriores, y aplicando los teoremas de coste, indicar el coste Temporal Asintótico del método.

$T^M_{contarMayoresQue}(x) \in \Theta(\log x)$ por Teorema 3, con a=1, c=2 y sobrecarga cte

→ **$T_{contarMayoresQue}(x) \in \Omega(\log x)$**

$T^P_{contarMayoresQue}(x) \in \Theta(x)$ por Teorema 4, con a=1, c= 2 (a<c) y sobrecarga lineal (por tamanyo)

→ **$T_{contarMayoresQue}(x) \in O(x)$**

v. Discutir en base al análisis de su coste si el (nuevo) método diseñado mejoraría su eficiencia si se dispusiera de un ABB con Rango (cuyos nodos contienen el atributo talla que representa su tamaño), en lugar de un simple ABB.

En un ABB con Rango, el cálculo del tamaño de un Nodo se realiza en el orden de una constante: basta acceder a su atributo talla. Por tanto, recordando que el código del (nuevo) método era ...

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

La búsqueda que representa tendría **en el caso general ...**

Caso Mejor: el dato e es igual al que ocupa el Nodo actual, por lo que en el orden de una constante se calcularía *tamanyo(actual.der)*.

Caso Peor: el dato e es distinto (menor o mayor) que el que ocupa el Nodo actual, por lo que habría que contar todos los nodos bien de su Hijo Izquierdo o bien de su hijo derecho.

Así que las Relaciones de recurrencia asociadas a estas instancias significativas serían:

$T^M_{contarMayoresQue}(x>0) = k_1$;

$T^P_{contarMayoresQue}(x>0) = 1 * T^P_{contarMayoresQue}(x/2) + k_2$;

Por lo que, aplicando los teoremas de coste, se tendrían los mismos resultados que para la versión eficiente del método *enqueNivel:* $T_{contarMayoresQue}(x) \in \Omega(1)$ y $T_{contarMayoresQue}(x) \in O(\log x)$

Ejercicio 3.- Modificando donde creas necesario el método **enQueNivel** diseñado en un ejercicio anterior, diseña en la clase **ABB** un método **hermanoDe** que, con coste mínimo, devuelva el dato en el Nodo Hermano del que contiene a uno **e** dado, o **null** si e no está en el ABB o no tiene hermano. Indica además el coste del método.

```
public E hermanoDe(E e){
  NodoABB<E> res = hermanoDe(x, raiz, null);
  if ( res != null )  return res.dato; else return null;
}
protected NodoABB<E> hermanoDe(E e, NodoABB<E> actual, NodoABB<E> hermano) {
  if ( actual==null ) return null;
  if ( actual.dato.compareTo(e)==0 ) return hermano;
  NodoABB<E>          res = hermanoDe(e, actual.izq, actual.der);
  if ( res == null )  res = hermanoDe(e, actual.der, actual.izq);
  return res;
}
```

```
protected NodoABB<E> hermanoDe(E e, NodoABB<E> actual, NodoABB<E> hermano) {
        if ( actual==null ) return null;
        int resC = actual.dato.compareTo(e);
        if ( resC==0 ) return hermano;
        if ( resC>0 )   return hermanoDe(e, actual.izq, actual.der);
        return hermanoDe(e, actual.der, actual.izq);
}
```

¿En que se parecen todos los ejercicios sobre ABBs al siguiente?

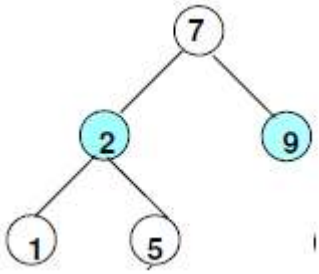
Último ejercicio propuesto en la transparencia nº 16 del Tema 2 (DyV).- Sea v un array de Integer positivos que se ajustan al perfil de una curva cóncava, i.e. existe una única posición k de v, 0≤k<v.length, tal que ∀j:0≤j<k:v[j]>v[j+1] (elementos ordenados descendentemente a la izquierda de k) y ∀j:k<j<v.length:v[j-1]<v[j] (elementos ordenados descendentemente a la izquierda de k). Por ejemplo:

			k						
4	3	2	1	2	3	4	5	6	7

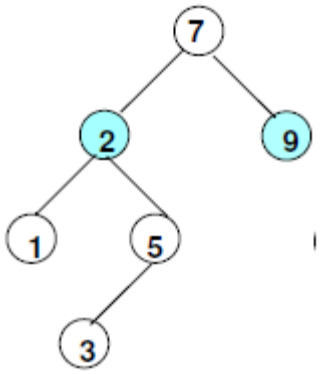
Diseñar el método recursivo que más eficientemente calcule tal posición k, la del mínimo de la curva.

```
public static int buscarPosK(Integer v[]){
   return buscarPosK(v, 0, v.length-1);
}
private static int buscarPosK(Integer v[], int inicio, int fin){
   if ( inicio>fin ) return -1;
   else {
        int mitad = ( inicio+fin)/2, resCAnt = 1, resCSig = 1; //por si no hay Anterior, o Posterior a mitad
        if ( mitad>inicio) resCAnt = v[mitad-1].compareTo(v[mitad]);
        if ( mitad<fin )   resCSig = v[mitad+1].compareTo(v[mitad]);
        // Si v[mitad-1]<v[mitad]<v[mitad+1] estamos en la parte creciente de la curva →
        // el mínimo no es v[mitad]  Y solo puede estar "antes" de mitad-1, en el subarray izq.
        if ( resCAnt<0 && resCSig>0 )          return buscarPosK(v, inicio, mitad–1);
        //Sino, si  v[mitad-1]>v[mitad]>v[mitad+1] estamos en la parte decreciente de la curva →
        // el mínimo no es v[mitad]  Y solo puede estar "después" de mitad+1, en el subarray der.
        else if  (resCAnt>0 && resCSig<0 )     return buscarPosK(v, mitad+1, fin);
        // Sino, tanto si no hay punto Anterior o Posterior a mitad como si v[mitad] es menor que
        // v[mitad-1] (el Anterior) Y v[mitad+1] (el Posterior), v[mitad] es el mínimo de la curva es
        else                                   return mitad;
}
Talla x = fin-inicio+1; Coste logarítmico con x.
```

Ejercicio 7.- Suponiendo que **no** se dispone del método **altura()**, **se pide** diseñar en la clase ABB un método **alturaDeEquilibrado** tal que si el ABB sobre el que se aplica es Equilibrado devuelve su altura y sino lanza la Excepción **ABBNoEquilibrado** tan pronto se encuentra un Nodo del ABB que incumple la condición de Equilibrio; por ejemplo, si **alturaDeEquilibrado** se aplica sobre el ABB de la siguiente figura entonces su resultado es 2.



Sin embargo, si se aplica sobre el ABB de la siguiente figura entonces lanza **ABBNoEquilibrado** con mensaje asociado "Encontrado Nodo Desequilibrado en el ABB".



Nota: se dice que un ABB es Equilibrado si la diferencia de alturas entre el Hijo Izquierdo y Derecho de cualquiera de sus Nodos es como máximo 1.

```
public int alturaDeEquilibrado()  throws ABBNoEquilibrado{
  return alturaDeEquilibrado(this.raiz);
}
protected int alturaDeEquilibrado(NodoABB<E> actual) throws ABBNoEquilibrado{
  if ( actual==null) return -1;
  else{
      int alturaIzq = alturaDeEquilibrado(actual.izq); int alturaDer = alturaDeEquilibrado(actual.der);
      int dif = Math.abs(alturaIzq - alturaDer);
      if ( dif>1 ) throw new ABBNoEquilibrado("Encontrado Nodo Desequilibrado en el ABB");
      return 1+Math.max(alturaIzq, alturaDer);
  }
}
```