Soluciones Ejercicios Tema 11

Germán Moltó Martínez
gmolto@dsic.upv.es
Estructuras de Datos y Algoritmos
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Versión Iterativa de recuperar en un ABB

```
protected NodoABB<E> recuperar (E clave, NodoABB<E> n)
   throws ElementoNoEncontrado{
       boolean esta = false;

       while ( n != null && !esta ){
        int resC =  n.dato.compareTo(clave);
        if (resC < 0) n = n.der;
        else if (resC > 0)  n = n.izq;
             else esta = true;
       }

    if (!esta)  throw new ElementoNoEncontrado("El elemento " +
   clave  + " no está");
    else return n;
 }
```


Versión iterativa de insertarConDuplicados

```
protected NodoABB<E> insertarConDuplicados(E clave, NodoABB<E> n){
    NodoABB<E> aux = n, padreAux = null;
    int resC = 0;
    while ( aux != null ) {
       resC = aux.dato.compareTo(clave); numTotalComparaciones++;
       padreAux = aux;
       if ( resC < 0 ) aux = aux.der; else aux = aux.izq;
      }
     NodoABB<E> nuevo = new NodoABB<E>(clave);
      if ( padreAux == null ) n = nuevo;
      else if ( resC < 0 ) padreAux.der = nuevo;
             else padreAux.izq = nuevo;
     numTotalInserciones++;
     return n;
 }
```


Versiones Alternativas

```
protected NodoABB<E> recuperarMax(NodoABB<E> n) {
    while (n.der != null) n = n.der;
    return n;
   }

protected NodoABB<E> recuperarMin(NodoABB<E> n){
   if (n.izq != null ) return recuperarMin(n.izq);
   else return n;
 }
```

Número de Hojas de ABB (1/3)

```
public class ABB<E extends Comparable<E>>{
  …
    public int numeroHojas(){
        return numeroHojas(raiz);
       }

    protected int numeroHojas(NodoABB<E> n){
       if ( n == null) return 0;
       else if (n.izq == null && n.der == null) return 1;
       else return numeroHojas(n.izq) + numeroHojas(n.der);
     }
    }
```


Número de Hojas de ABB (2/3)

- Notación: numHojas(i) significa la invocación al método numHojas pasándole como argumento la referencia al NodoABB<E> que contiene el objeto i.

- Órden en el que se producen las llamadas recursivas:

numHojas(10) → numHojas(5) → numHojas(4) → numHojas(6) → numHojas(11) → numHojas(null) → numHojas(15)

- Órden en el que finalizan las llamadas recursivas:

numHojas(4) → numHojas(6) → numHojas(5) → numHojas(null) → numHojas(15) → numHojas(11) → numHojas(10)

Número de Hojas de ABB (3/3)

- Estilo de implementación alternativo, donde el caso base es tener una hoja.

```
public int numeroHojas(){
   if (raiz != null) return numeroHojas(raiz);
   else return 0;
 }

protected int numeroHojas(NodoABB<E> n){
    int nHojas = 0;
    if (n.izq == null && n.der == null) nHojas = 1;
    else {
      if (n.izq != null) nHojas += numeroHojas(n.izq);
      if (n.der != null) nHojas += numeroHojas(n.der);
      }
    return nHojas;
   }
```


Borrar Hojas de ABB

```
public class ABB<E extends Comparable<E>>{
 …
   public void borrarHojas(){
        if ( this.raiz != null ) this.raiz = borrarHojas(this.raiz);
     }

   protected NodoABB<E> borrarHojas(NodoABB<E> n){
         NodoABB<E> res = n;
         if ( n.izq == null   &&  n.der == null ) res = null;
         else{
               if ( n.izq != null )  n.izq =  borrarHojas(n.izq);
               if ( n.der != null ) n.der =  borrarHojas(n.der);
      }
      return res;
   }}
```

Esfuerzo Medio de Comparación Óptimo

```
public double eMCOptimo(){
    double res = Double.NaN;
     if ( !seHaEliminado ){
        double log2Tamanyo = Math.log(this.tamanyo())/Math.log(2);
        long alturaOptima = Math.round(Math.floor(log2Tamanyo));
        long valor = 1, aux = 1, numNodos=1;
        for (int i = 1; i <= alturaOptima ; i++){
              aux = aux*2;   numNodos += aux;
              valor += aux * (i+1);
           }
           res = ((double)valor)/numNodos;
        }
        return res; }
```


Mostrar los Nodos de un Cierto Nivel

```
//SII 0 <= k <=altura():
public String toStringNivel(int k){
      if ( this.raiz != null )  return toStringNivel(k, this.raiz);
      else return "*";
   }
protected String toStringNivel(int k, NodoABB<E> n){
       String res = "";
       if ( k == 0 ) res += n.dato.toString()+" "; // Caso base
       else {
           if( n.izq != null ) res += toStringNivel(k-1, n.izq);
           if( n.der != null ) res += toStringNivel(k-1, n.der);
       }
       return res;
    }
```


Suma de Elementos Mayores o Igual

```
public class ABBInteger extends ABB<Integer>{
 public int sumarMayorOIgual(int x) {
     return sumarMayorOIgual(this.raiz, x);
}
protected int sumarMayorOIgual(NodoABB<Integer> n, int x) {
   int suma=0;
   if (n!=null) {
       int valorRaiz= n.dato.intValue();
       if (valorRaiz>=x) suma +=valorRaiz;
       if (valorRaiz>x) suma += sumarMayorOIgual(n.izq, x);
       suma += sumarMayorOIgual(n.der,x);
    }
    return suma;
}
```


Construcción ABB Equilibrado

- Sea N = v.length el número de elementos de v. Al estar ordenados y ser distintos entre sí, su elemento central es mayor que los (N/2) - 1 elementos anteriores y menor que los (N/2) – 1 elementos posteriores.
- La propiedad se puede aplicar recursivamente a los subvectores resultantes.

```
private NodoABB<E> seConstruye(E v[], int izq, int der){
   NodoABB<E> nuevo = null;
   if ( izq <= der ){
       int centro = (izq+der)/2;
       nuevo = new NodoABB<E>(v[centro]);
       nuevo.izq = seConstruye(v, izq, centro-1);
       nuevo.der = seConstruye(v, centro+1, der);
   }
   return nuevo;
}
```

Construcción ABB Equilibrado (II)

- Complejidad temporal del método seConstruye:
- Talla del problema: Viene dada por la expresión n = der – izq + 1. En la llamada más alta coincide con v.length.
- Instancias Significativas: NO existen, puesto que se trata de un recorrido recursivo. Siempre se visitarán todas las componentes del vector.
- Relaciones de recurrencia:
  - $T_{seConstruye}(n>0) = 2 * T_{seConstruye}(n/2) + k$
  - $T_{seConstruye}(n=0) = k'$
- Acotando con el Teorema 3 (a = 2, c = 2):

$T_{seConstruye} \in \Theta(v.length)$

Cálculo del Sucesor

```
protected NodoABB<E> sucesor(E clave,NodoABB<E> n)  throws
  ElementoNoEncontrado  {
   NodoABB<E> ascenDer=null;   boolean encontradaClave=false;
    while ( n!=null && !encontradaClave){
      int resC = n.dato.compareTo(clave);
      if (resC<0) {n=n.der;}
      else if (resC>0) {ascenDer = n; n = n.izq;}
            else encontradaClave=true;
    }
   if (!encontradaClave ) throw new ElementoNoEncontrado("No esta");
   else if (n.der !=null) return recuperarMin(n.der);
         else return ascenDer;
  }
```


toString en ABBColaPrioridad

- Aprovechamos el conocimiento de que un recorrido en inorden de un Árbol Binario de Búsqueda obtiene los elementos en orden ascendente.

```
public String toString(){
    return toStringInOrden();
   }
```


Cambia Signo Árbol Binario de Búsqueda

```
public class ABBInteger extends ABB<Integer>{

public void cambiarSigno() {
   cambiarSigno(raiz);
}
protected void cambiarSigno(NodoABB<Integer> n) {
   if (n != null) {
          int dato  = n.dato.intValue();
          n.dato = new Integer(-dato);
          NodoABB<Integer> aux = n.izq;
          n.izq = n.der;
          n.der = aux;
          cambiarSigno(n.izq);
          cambiarSigno(n.der);
   }
 }
```

Padre de un Nodo en un ABB

```
public NodoABB<E> padre(E x, NodoABB<E> n){
    NodoABB<E> p = null;
    boolean esta = false;
    while (n!= null && !esta){
      int resC = n.dato.compareTo(x);
      if (resC>0) {p = n; n = n.izq;}
      else if (resC<0) {p = n; n = n.der;}
           else esta=true;
    }
    if (esta) return p;
    else return null; }
```


Fuera de Intervalo (1/2)

```
public int fueraDeRango(E x, E y) {
   return fueraDeRango(x,y, this.raiz);
}
```


Fuera de Intervalo (2/2)

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


Coste Temporal de Contar Mayores (I)

- Talla del Problema en función de los argumentos:
  - Talla = actual.tamanyo (o tamanyo(actual))
- ¿Instancias Significativas?
  - No las hay, es un problema de recorrido que tiene que procesar todos los nodos del ABB.
- Relaciones de recurrencia para un ABB equilibrado:
  - $T_{contarMayoresQue}(x = 0) = k2$
  - $T_{contarMayoresQue}(x > 0) = 2 * T_{contarMayoresQue}\ (x/2) + k3$
- Relaciones de recurrencia para un ABB degenerado:
  - $T_{contarMayoresQue}(x = 0) = k2$
  - $T_{contarMayoresQue}(x > 0) = 1 * T_{comparar}(x-1) + k3$

Coste Temporal de Contar Mayores (II)

- Coste temporal asintótico:
  - Si actual es un nodo de un ABB completamente degenerado: por Teorema 1 con a = c =1 y sobrecarga constante:
    - $T_{contarMayoresQue}(x) \in \Theta(x)$
  - Si actual es un nodo de un ABB equilibrado: por Teorema 3 con a = c = 2 y sobrecarga constante:
    - $T_{contarMayoresQue}(x) \in \Theta(x)$
- Se obtienen las mismas cotas porque es un problema de recorrido y, al final, hay que procesar todos los nodos.

Solución Eficiente a Contar Mayores

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


toString Ordenado Ascendentemente

```
public String  toStringOAMayoresQue(E e) throws ElementoNoEncontrado{
  String res=toStringOAMayoresQue(e, this.raiz);
   if ( res.equals("") )  throw new ElementoNoEncontrado("No existe");
   return res;
 }
protected String toStringOAMayoresQue (E e, NodoABB<E> actual){
  String res = "";
  if ( actual != null ){
  if ( actual.dato.compareTo(e) > 0 ){
       res+=toStringOAMayoresQue (e, actual.izq) + actual.dato.toString();}
   res+=toStringOAMayoresQue(e, actual.der);
 }
 return res;  }
```


Mediana en ABB

```
public boolean esMediana(E x) {
NodoABB<E> aux = this.raiz; int mayores = 0, int menores = 0;
boolean fin = false;
while ( aux !=null && !fin ) {
  int resC = aux.dato.compareTo(x);
  if (resC < 0) { menores += 1 + tamanyo(aux.izq); aux = aux.der; }
  else if (resC > 0) { mayores += 1 + tamanyo(aux.der); aux = aux.izq; }
        else {
           menores += tamanyo(aux.izq); mayores += tamanyo(aux.der);
           fin = true;
       }
}
return (mayores == menores);}
```

Altura de Equilibrado

```
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 diff = Math.abs(alturaIzq - alturaDer);
        if ( diff > 1 ) throw new ABBNoEquilibrado("Nodo Desequilibrado ");
        return 1+Math.max(alturaIzq, alturaDer);
    } }
```


toString en Rango (I)

1. Ecuaciones de Recurrencia si es un Nodo Equilibrado
   - $T_{toStringRango}(x = 0) = k2$
   - $T_{toStringRango}(x > 0) = 2 * T_{toStringRango}(x/2) + k3$
2. Relaciones de Recurrencia si es Nodo degenerado
   - $T_{toStringRango}(x = 0) = k2$
   - $T_{toStringRango}(x > 0) = 1 * T_{toStringRango}(x-1) + k3$
3. Cotas de Complejidad Temporal Asintótica
   - En ambos casos: $T_{toStringRango}(x) \in \Theta(x)$

- Se trata de un problema de recorrido, no hay instancias significativas y el coste es lineal con el número de elementos.

toString en Rango (II)

```
protected String toStringEnRango(NodoABB<E> actual, E inf, E sup){
  String res = "";
  if ( actual != null ) {
    int rInf = actual.dato.compareTo(inf);
    int rSup = actual.dato.compareTo(sup);
    if ( rInf < 0 ) res += toStringEnRango(actual.der, inf, sup);
    else if ( rSup > 0 ) res += toStringEnRango(actual.izq, inf, sup);
        else { // inf <= dato <= sup
        res += toStringEnRango(actual.izq, inf, sup);
        res += actual.dato.toString()+"\n";
        res += toStringEnRango(actual.der, inf, sup);
  }
  return res; }
```