Estructuras de Datos Avanzadas

Tarea 2: Arboles Binarios de Búsqueda

Prof.: Fernando Esponda

Alumno: Guillermo Arredondo Renero

C.U.: 000197256

Experimento para medición de comparaciones por parte de Inserción y Eliminación de Datos en Árboles Binarios de Búsqueda

Introducción:

Los árboles binarios de búsqueda son estructuras de datos naturalmente recursivas que cuentan sirven para el almacenamiento de datos ordenados a través de nodos conectados 2 a 1. De este modo, cada nodo cuenta con solo 2 hijos como máximo. Es importante notar que este tipo de árboles no mantienen una relación en cuanto al orden presentado, sino que agregan los datos conforme el orden en que se les presentan, acomodándolos únicamente con base en el criterio de menor, igual o mayor que el nodo anterior. De este modo los árboles binarios de búsqueda cumplen:

> Todos los elementos menores o iguales que un nodo base se encuentran a su izquierda
>
> Todos los elementos mayores que un nodo base se encuentran a su derecha

Experimento:

Se diseñó un experimento basado en 3 casos principales de acuerdo al orden en que se acomodan los datos del árbol binario de búsqueda:

1. Caso aleatorio, en el cual se representa el caso medio, para ello se utilizan ciclos que rellenan el árbol a partir de elementos integer aleatorios.
2. Caso ordenado, en el cual se ingresan datos ordenados de forma ascendente (sin pérdida de generalidad), representando el peor caso posible de eficiencia en un árbol binario, puesto que representan a una lista.
3. Caso de un árbol balanceado, el mejor de los casos en un árbol, pues la inserción y búsqueda siempre tarda log n.

Para medir el número de comparaciones que se llevan a cabo en los métodos de inserción y eliminación se alteraron los métodos originales de inserción y eliminación para que regresaran variables enteras contando el número de preguntas if que se hicieran. Para ello se usaron variables contadoras auxiliares. Es importante aclarar que se tomaron como comparaciones las preguntas de valoración de un nodo distinto o igual a nulo por igual que comparaciones entre datos, ya que de todas formas en el conteo no refleja una brecha mayor que n pasos.

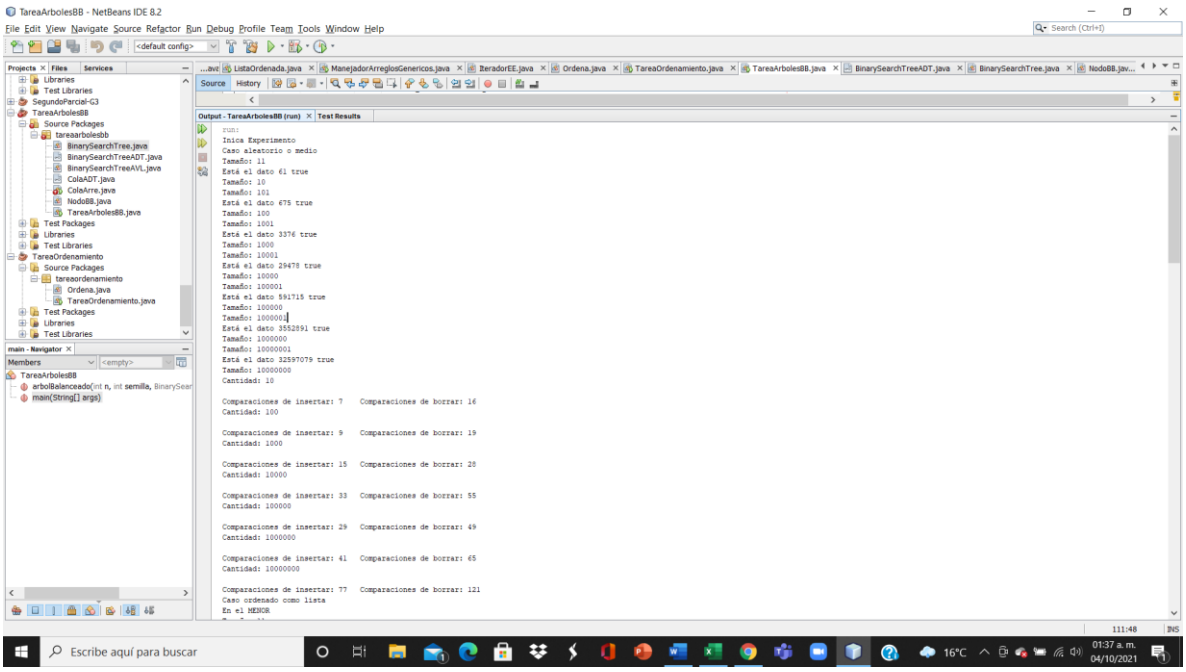
Para presentar los casos de distintas cantidades de datos se utilizaron una serie de ciclos y switch’s para organizar todos los casos y obtener cada cantidad de comparaciones por cantidad de datos. Para ello se usaron dos listas, una que almacenara la cantidad de datos y otra que almacenara las comparaciones, primero las de inserción y después las de eliminación.

Para el caso de generar árboles binarios balanceados se procedió con el siguiente algoritmo:

```
public static void arbolBalanceado(int n, int semilla /*Math.pow(2,n-1)*/, BinarySearchTreeADT<Integer> arbol, int i/*2*/){
    if(i-1<=n){
        arbol.add(semilla);
        arbolBalanceado(n,semilla+(int)Math.pow(2, n-i),arbol,i+1);
        arbolBalanceado(n,semilla-(int)Math.pow(2,n-i),arbol,i+1);
    }
}
```

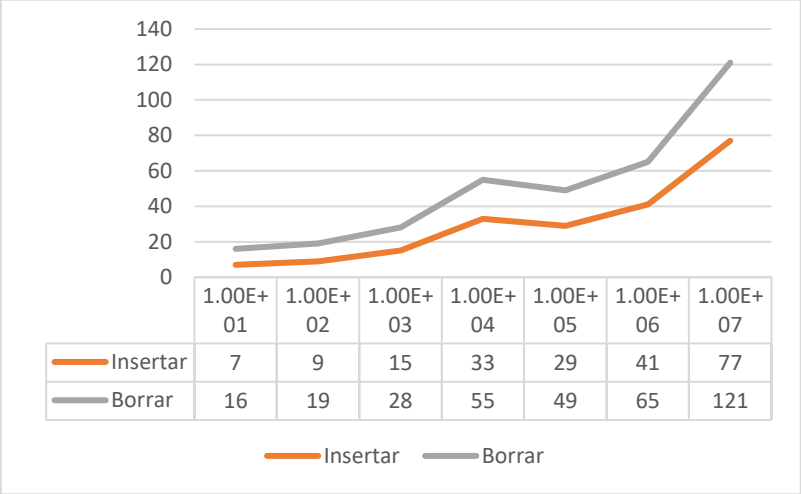
Como se puede ver, se genera un árbol binario de 2^n-1 elementos, por lo tanto, para obtener lo más aproximado a n elementos (para 100, 1000, 10000, etc) es necesario hacer la conversión a través de logaritmos como sigue: arbolBalanceado((int)Math.floor(Math.log(n+1)/Math.log(2)),(int)Math.pow(2,Math.log(n+1)/Math.log(2)),arbol, 2);

Resultados

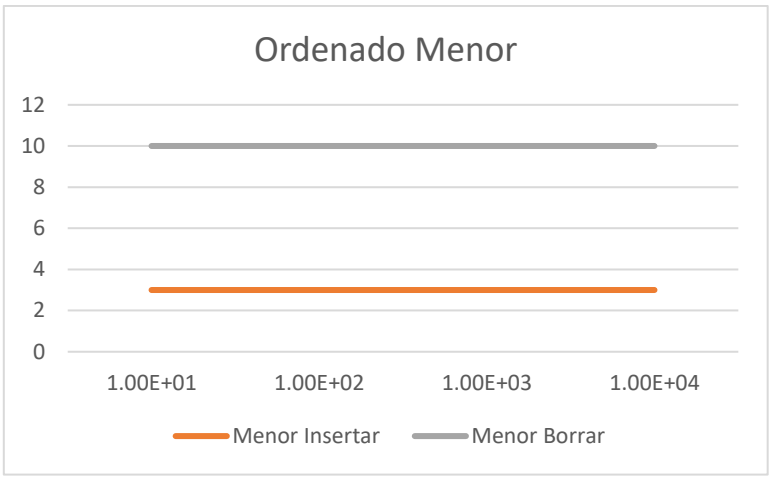


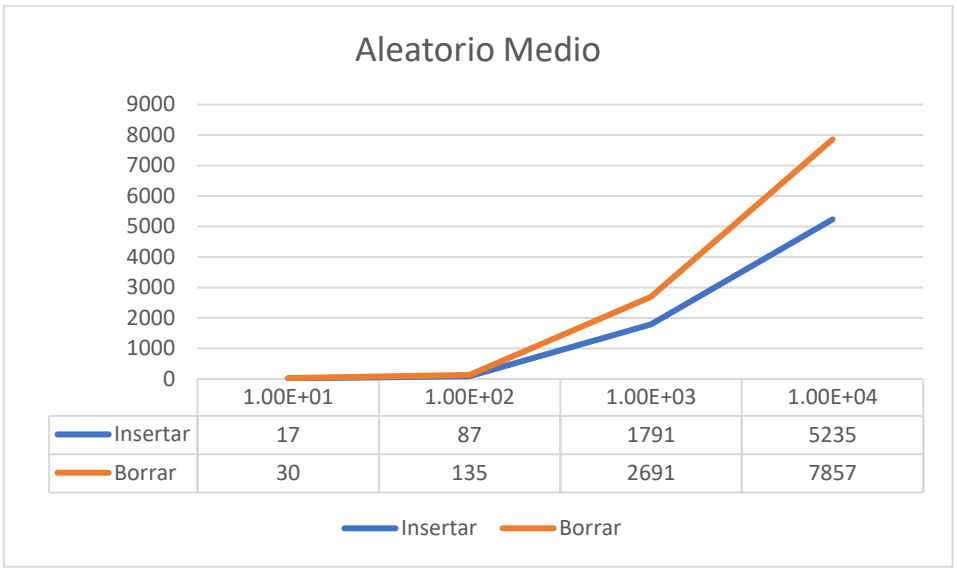
Aleatorio

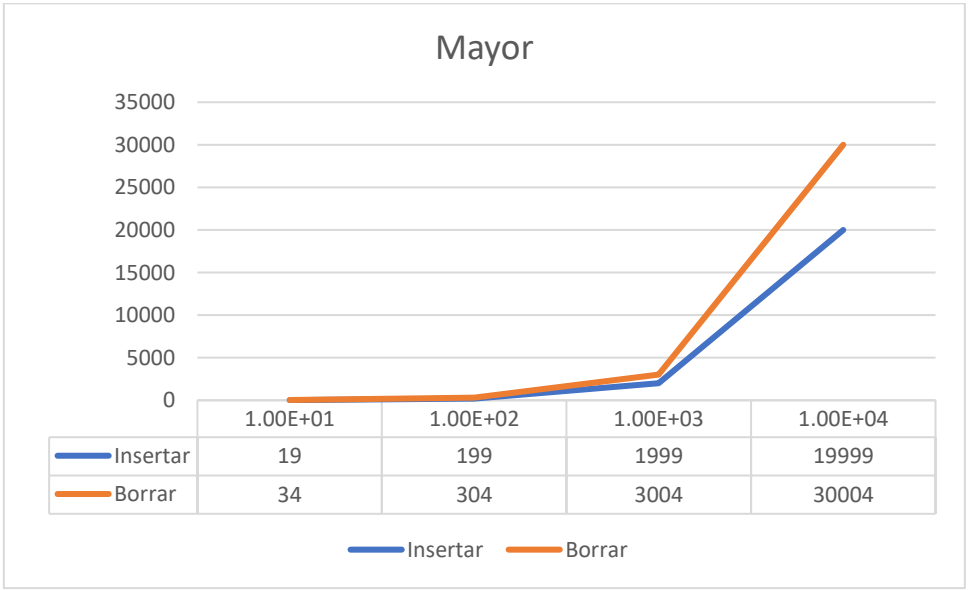
Cantidades de Datos	Insertar	Borrar
1.00E+01	7	16
1.00E+02	9	19
1.00E+03	15	28
1.00E+04	33	55
1.00E+05	29	49
1.00E+06	41	65
1.00E+07	77	121



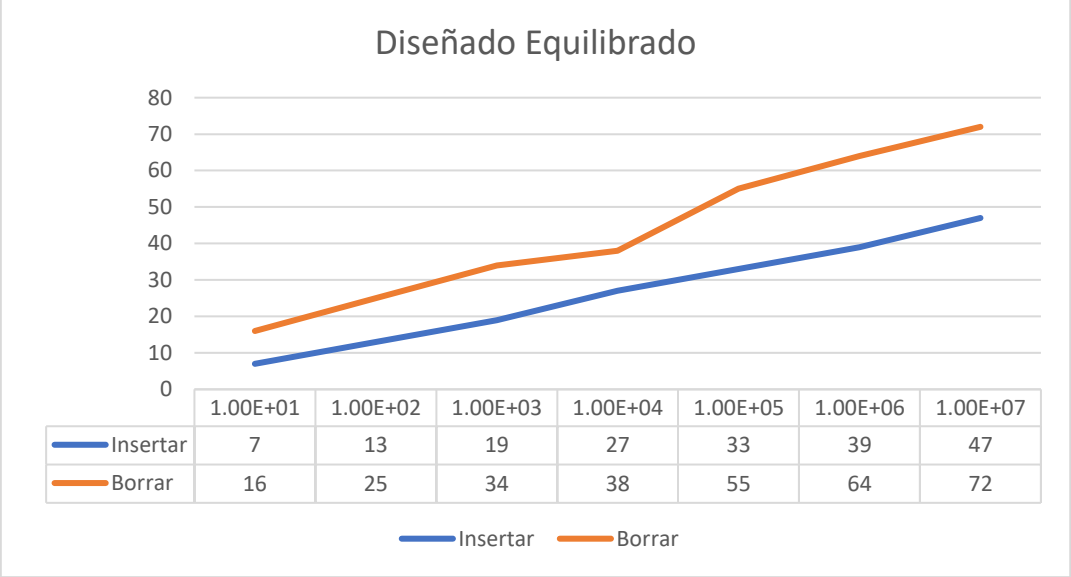
Ordenado						
Caso	**Menor**		**Aleatorio Medio**		**Mayor**	
Cantidades de Datos	Insertar	Borrar	Insertar	Borrar	Insertar	Borrar
1.00E+01	3	10	17	30	19	34
1.00E+02	3	10	87	135	199	304
1.00E+03	3	10	1791	2691	1999	3004
1.00E+04	3	10	5235	7857	19999	30004

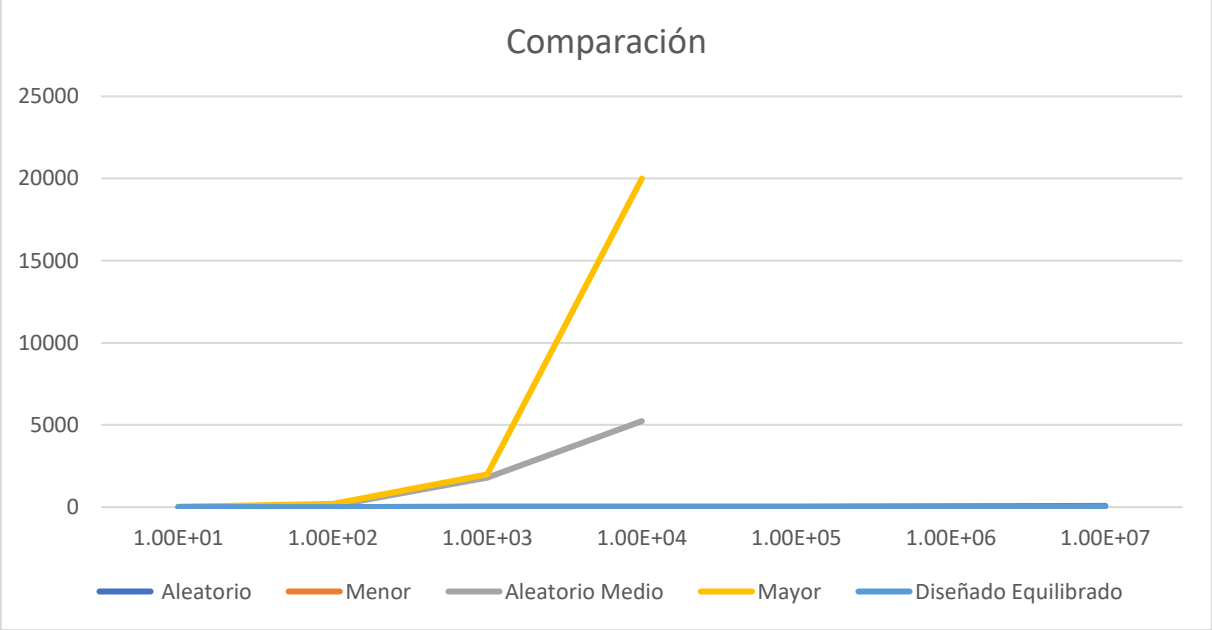

Ordenado Menor
12
10
8
6
4
2
0
1.00E+01
1.00E+02
1.00E+03
1.00E+04
Menor Insertar
Menor Borrar


Aleatorio Medio
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
1.00E+01
1.00E+02
1.00E+03
1.00E+04
Insertar
17
87
1791
5235
Borrar
30
135
2691
7857
Insertar
Borrar


Mayor
35000
30000
25000
20000
15000
10000
5000
0
1.00E+01
1.00E+02
1.00E+03
1.00E+04
Insertar
19
199
1999
19999
Borrar
34
304
3004
30004
Insertar
Borrar

Diseñado Equilibrado		
Cantidades de Datos	Insertar	Borrar
1.00E+01	7	16
1.00E+02	13	25
1.00E+03	19	34
1.00E+04	27	38
1.00E+05	33	55
1.00E+06	39	64
1.00E+07	47	72





Conclusiones

Puede verse que los resultados concuerdan con los conceptos teóricos estudiados y esperados. Por un lado, observamos que en el caso medio o aleatorio las comparaciones toman valores muy similares al del mejor de los casos; sin embargo, es importante notar que esto no siempre puede ocurrir así, pues es muy complicado que un árbol se llene perfectamente o se encuentre balanceado. En caso de conseguirse, la complejidad y cantidad de comparaciones siempre será de O(logn). Asimismo, el peor de los casos, ampliamente conocido presenta una complejidad mucho mayor, una vez que se trata de llevar alrededor de n comparaciones e incluso varias más.

Por último, es importante rescatar que el caso del método de eliminación lleva a cabo en la mayoría de los casos aproximadamente el doble de comparaciones, esto se debe a la necesidad de realizar múltiples consideraciones de casos de nodos a eliminar y de la necesidad de reordenar el árbol, así como que la mayor cantidad de comparaciones surgen del método de búsqueda.

Apéndice:

Código utilizado para pruebas unitarias

```
/**
 *
 * @author Guillermo Arredondo
 */
public class TareaArbolesBB {
    public static void arbolBalanceado(int n, int semilla /*0*/, BinarySearchTreeADT<Integer> arbol, int i/*2*/){
        if(i-1<=n){
            arbol.add(semilla);
            arbolBalanceado(n,semilla+(int)Math.pow(2, n-i),arbol,i+1);
            arbolBalanceado(n,semilla-(int)Math.pow(2,n-i),arbol,i+1);
        }
    }
public static void main(String[] args) {
        BinarySearchTreeADT<Integer> arbol = new BinarySearchTree();
        Random rand = new Random();
        int nuevo, n0=4;
        /*arbolBalanceado(n0,(int)Math.pow(2,n0 -1),arbol,2);
        Iterator<Integer> it = arbol.inOrden();
        while(it.hasNext()){
            System.out.println(it.next());
        }
        /*System.out.println("Vacio: " + arbol.isEmpty());
        arbol.add(1);
        for(int i = 0; i<10; i++){
            nuevo = rand.nextInt(50);
            System.out.print(nuevo+" ");
```

```
    arbol.add(nuevo);
}
nuevo = 34;
arbol.add(nuevo);
System.out.println("Vacio?: " + arbol.isEmpty());
System.out.println("Tamaño: "+ arbol.size());
System.out.println("Está el dato 34? " + arbol.find(nuevo));
System.out.println("Inorden");
Iterator<Integer> it = arbol.inOrden();
while(it.hasNext()){
    System.out.println(it.next());
}
System.out.println("Preorden");
it = arbol.preOrden();
while(it.hasNext()){
    System.out.println(it.next());
}
System.out.println("PostOrden");
it = arbol.postOrden();
while(it.hasNext()){
    System.out.println(it.next());
}
System.out.println("hasta ahora todo funciona perfecto");
arbol.borra(nuevo);
System.out.println("Tamaño: "+ arbol.size());
System.out.println("Inorden");
it = arbol.inOrden();
while(it.hasNext()){
    System.out.println(it.next());
```

```
}
System.out.println("Pruebas unitarias de métodos funcionan adecuadamente");


System.out.println("Pruebas para experimento");
System.out.println("Insertar: "+arbol.add(50));
System.out.println("Borrar: "+arbol.borra(1));
*/
```

Experimento

```
System.out.println("Inica Experimento");
ArrayList<Integer> cantDatos= new ArrayList();
ArrayList<Integer> numComps=new ArrayList();
int contCompsInsert, contCompsBorra,k;
for (int tipo = 1; tipo <= 3; tipo++) {
   switch (tipo) {
      case 1:
         System.out.println("Caso aleatorio o medio");
         for (int n = 10; n <= 10000000; n *= 10) {
            cantDatos.add(n);
            contCompsInsert = 0;
            contCompsBorra = 0;
            for (int i = 0; i < n; i++) {
               nuevo = rand.nextInt(n * 5);
               arbol.add(nuevo);
            }
            nuevo = rand.nextInt(n * 7);
            numComps.add(arbol.add(nuevo));
            System.out.println("Tamaño: " + arbol.size());
            System.out.println("Está el dato " + nuevo + " " + arbol.find(nuevo));
            numComps.add(arbol.borra(nuevo));
```

```
            System.out.println("Tamaño: " + arbol.size());
            arbol = new BinarySearchTree();
        }
        for (int j = 0; j < cantDatos.size(); j++) {
            k = 2 * j + 1;
            System.out.println("Cantidad: " + cantDatos.get(j));
            System.out.print("\nComparaciones de insertar: " + numComps.get(k - 1));
            System.out.println("\tComparaciones de borrar: " + numComps.get(k));
        }
        cantDatos.clear();
        numComps.clear();
            break;
    case 2:
        System.out.println("Caso ordenado como lista");
        for (int n = 10; n <= 10000; n *= 10) {
            contCompsInsert = 0;
            contCompsBorra = 0;
            for (int i = 0; i < n; i++) {
                arbol.add(i);
            }
            for(int caso=1; caso<=3; caso++){
                switch(caso){
                    case 1:
                        System.out.println("En el MENOR");
                        nuevo=-8;
                        break;
                    case 2:
                        System.out.println("Aleatorio en medio");
                        nuevo=rand.nextInt(n-1);
```

```
                    break;
                default:
                    System.out.println("En el MAYOR");
                    nuevo=n+4;
                    break;
                }
                contCompsInsert = arbol.add(nuevo);
                System.out.println("Tamaño: " + arbol.size());
                System.out.println("Está el dato " + nuevo + " " + arbol.find(nuevo));
                contCompsBorra = arbol.borra(nuevo);;
                System.out.println("Tamaño: " + arbol.size());
                System.out.println("Cantidad: " + n);
                System.out.print("\nComparaciones de insertar: " + contCompsInsert);
                System.out.println("\tComparaciones de borrar: " + contCompsBorra);
            }
            arbol = new BinarySearchTree();
        }
        cantDatos.clear();
        numComps.clear();
            break;
        default:
            System.out.println("Arbol diseñado y equilibrado *no por fuerza lleno*");
            for (int n = 10; n <= 10000000; n *= 10) {
                cantDatos.add(n);
                contCompsInsert = 0;
                contCompsBorra = 0;
                arbolBalanceado((int)Math.floor(Math.log(n+1)/Math.log(2)),(int)Math.pow(2,
Math.log(n+1)/Math.log(2)),arbol, 2);
                nuevo = rand.nextInt(n*3);
```

```
                numComps.add(arbol.add(nuevo));
                System.out.println("Tamaño: " + arbol.size());
                System.out.println("Está el dato " + nuevo + " " + arbol.find(nuevo));
                numComps.add(arbol.borra(nuevo));
                System.out.println("Tamaño: " + arbol.size());
                arbol = new BinarySearchTree();
            }
            for (int j = 0; j < cantDatos.size(); j++) {
                k = 2 * j + 1;
                System.out.println("Cantidad: " + cantDatos.get(j));
                System.out.print("\nComparaciones de insertar: " + numComps.get(k - 1));
                System.out.println("\tComparaciones de borrar: " + numComps.get(k));
            }
            cantDatos.clear();
            numComps.clear();
            break;
        }
//          System.out.println("Inorden");
//          it = arbol.inOrden();
//          while (it.hasNext()) {
//              System.out.println(it.next());
//          }
//          System.out.println("Inorden");
//          it = arbol.inOrden();
//          while(it.hasNext()){
//              System.out.println(it.next());
//          }
    }
```

Métodos Utilizados

Busca

```
private void cuentaBusca(T elem, Object[] res){
        NodoBB<T> hoja = raiz;
        int contAux = 2;
        while(hoja!=null && !hoja.elem.equals(elem)){
            contAux+=3;
           if (hoja.elem.compareTo(elem) > 0)
              hoja=hoja.izq;
            else
              hoja=hoja.der;
        }
        if(hoja==null)
           contAux--;
        res[0]=hoja;
        res[1]= contAux;
    }
```

ADD

```
public int add(T elem) {
        NodoBB<T> nuevo = new NodoBB(elem);
        int contComps = 1;
        if (this.isEmpty()) {
           raiz = nuevo;
        } else {
           contComps += add(nuevo, raiz, 0);
        }
        cont++;
        return contComps;
```

```
    }
    private int add(NodoBB<T> nuevo, NodoBB<T> actual, int contComps) {
        contComps++;
        if (nuevo.getDato().compareTo(actual.getDato()) <= 0) {
            contComps++;
            if (actual.izq == null) {
                actual.setIzq(nuevo);
            } else {
                return contComps + add(nuevo, actual.izq, 0);
            }
        } else {
            contComps++;
            if (actual.der == null) {
                actual.setDer(nuevo);
            } else {
                return contComps + add(nuevo, actual.der, 0);
            }
        }
        return contComps;
    }
```

Borra

```
public int borra(T elem) {
        Object[] res = new Object[2];
        cuentaBusca(elem, res);
        NodoBB<T> actual = (NodoBB<T>)res[0];
        int contAux=1+(Integer)res[1];
        if (actual == null) {
            return contAux;
        }
```

```
    contAux+=3;
    if (actual.der == null && actual.izq == null) {//caso hoja
        contAux--;
        contAux+= eliminaHoja(actual);
    } else if (actual.der == null) {//un solo hijo
        contAux+= eliminaConHijoIzq(actual);
    } else if (actual.izq == null) {
        contAux+= eliminaConHijoDer(actual);
    } else { //dos hijos
        contAux+= eliminaDosHijos(actual);
    }
    cont--;
    return contAux;
}

private int eliminaHoja(NodoBB<T> actual) {
    if (actual == raiz) {
        raiz = null;
        return 1;
    }
    if (actual.papa.der == actual) {
        actual.papa.setDerNull();
    } else {
        actual.papa.setIzqNull();
    }
    return 2;
}

private int eliminaConHijoDer(NodoBB<T> actual) {
```

```
    NodoBB<T> hijo = actual.der;
    if (actual == raiz) {
        raiz = hijo;
        actual.setDerNull();
        raiz.papa=null;
        return 1;
    }
    else{
        actual.papa.cuelga(hijo);
        return 2;
    }
}

private int eliminaConHijoIzq(NodoBB<T> actual) {
    NodoBB<T> hijo = actual.izq;
    if (actual == raiz) {
        raiz = hijo;
        raiz.papa=null;
        actual.setIzqNull();
        return 1;
    }else{
        actual.papa.cuelga(hijo);
        return 2;
    }
}

private int eliminaDosHijos(NodoBB<T> actual) {
    NodoBB<T> sucInorden = actual.getDer();
    int contAux=1;
```

```
    while (sucInorden.getIzq() != null) {
        contAux++;
        sucInorden = sucInorden.getIzq();
    }
    actual.setDato(sucInorden.getDato());
    contAux++;
    if (actual.getDer() == sucInorden) {//caso en que no avanza
        actual.der = sucInorden.der;
        contAux++;
        if (actual.der != null) {
            actual.der.papa = actual;
        }
    } else {
        sucInorden.papa.izq = sucInorden.der;
        contAux++;
        if (sucInorden.der != null) {
            sucInorden.der.papa = sucInorden.papa;
        }
    }
    return contAux;
}
```