Estructuras de Datos y Algoritmos I
2º Grado en Ingeniería Informática

ACTIVIDAD 2

Implementación de operaciones sobre ABBs

Objetivos

- Aprender a utilizar árboles binarios de búsqueda (ABB) para tener un conjunto de datos ordenados según una clave y una relación de orden.
- Conocer el fundamento y uso de ABB balanceados como son el AVL y el Árbol Rojo-Negro. Realizar comparativas con ellos con el objetivo de ver cuál es el más apropiado dependiendo del caso.
- Aprender a añadir funcionalidad adicional a una estructura básica tipo ABB.

Requisitos

Para superar esta actividad se debe:

- Dominar el proceso de incrementar la funcionalidad de estructuras arborescentes.
- Dominar el proceso de comparación de distintas estructuras de datos arborescentes como son: ABB, AVL y Rojo-Negro…extrayendo las conclusiones correspondientes acorde con lo aprendido en teoría.

Enunciado

En esta actividad vamos a complementar lo estudiado en las clases de teoría con la utilización de Árboles Binarios de Búsqueda (`BSTree<T>`) y ABB balanceados tipo AVL (`AVLTree<T>`) y Rojo-Negros (`RBTree<T>`). Para ello, se hará uso de las implementaciones base suministradas en el repositorio…que se deberán extender/ampliar convenientemente con la funcionalidad que se indica.

Ejercicio 1. Implementad un nuevo método en la clase **`BSTree<T>`** que calcule la altura del árbol, en su versión recursiva.

```
private int height(BSTNode<T> current) {
   //…
}

public int height() {
   return height(root);
}
```

Ejercicio 2. Implementad un nuevo método en la clase **`BSTree<T>`** que obtenga el número total de hojas que tiene el árbol, en su versión recursiva.

```
private int numberOfLeaves(BSTNode<T> current) {
   //…
}
```

```
public int numberOfLeaves() {
   return numberOfLeaves(root);
}
```

Ejercicio 3. Implementad dos nuevos métodos en la clase **`BSTree<T>`** que encuentre el mínimo y el máximo de los elementos que tiene el árbol (si no está vacío), tanto en su versión recursiva como iterativa.

```
private BSTNode<T> findMin(BSTNode<T> current) {
   //…
}
public T findMin() {
   if (this.root == null) return null;
   return findMin(root).nodeValue;
}

public T findMinIterative() {
   //…
}

private BSTNode<T> findMax(BSTNode<T> current) {
   //…
}

public T findMax() {
   if (this.root == null) return null;
   return findMax(root).nodeValue;
}

public T findMaxIterative() {
   //…
}
```

Ejercicio 4. Implementad una nueva función (recursiva) en la clase **`BSTree<T>`** que devuelva la información de los nodos que están en un determinado nivel (proporcionado como parámetro).

```
private String toStringLevel(BSTNode<T> t, int currentLevel, int level){
   //…
}

public String toStringLevel(int level){
   return toStringLevel(root, 0, level);
}
```

Ejercicio 5. Implementad una nueva función para la clase **`BSTree<T>`** que devuelva la profundidad desde la raíz a la que se encuentra un nodo que contiene un determinado valor de clave **item**. Si dicho valor no se encuentra en el árbol, la función deberá devolver -1. La implementación deberá ser recursiva.

```
private int pathHeight(BSTNode<T> current, T item, int altura){
   //…
}

public int pathHeight(T item){
    return pathHeight(root, item, 0);
}
```

Ejercicio 6. Implementad tanto en el `AVLTree<T>` como en el `RBTree<T>` (ya está implementada en el ejercicio anterior para `BSTree<T>`) la misma función `pathHeight()`, que devuelve la profundidad desde la raíz a la que se encuentra un nodo que contiene un determinado valor de clave **item**. Si dicho valor no se encuentra en el árbol, la función deberá devolver -1. La implementación de dicha función, como ya se indicó en el ejercicio anterior, deberá ser recursiva.

Ejercicio 7. Implementad un método `code()` (recursivo) que codifique en binario una secuencia de valores a partir de su posición en un ABB. La lógica a seguir es algo similar a la utilizada por el algoritmo de *Huffman* para la compresión de datos. Empezando por el nodo raíz, asociado con el código 0, los nodos se irán codificando según el nivel en el que se encuentre, añadiendo un 0 si está situado a la izquierda, o un 1 si está situado a la derecha del nodo padre. Por ejemplo, si tenemos en cuenta el árbol que se muestra en la siguiente figura, el valor 40 estaría asociado con el código 0, ya que se trata del nodo raíz, mientras que el nodo que contiene el valor 45 estaría asociado con el código 010. Así pues, la frase 40 10 50 se codificaría como "0 000 01". A continuación, vamos a hacer una implementación iterativa del método `decode()` para decodificar una frase y obtener la secuencia de elementos original.

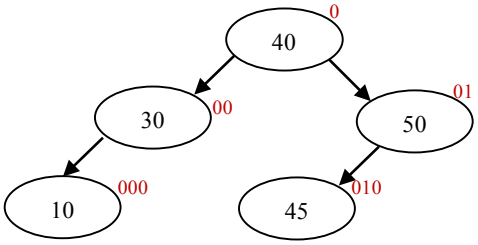
```
private String code(BSTNode<T> current, T item, String codigo){
   //…
}

public String code(ArrayList<T> frase){
   String result="";
   for (int i=0; i<frase.size(); i++)
      result += code(root, frase.get(i),"0") + " ";
   return result;
}

private T decode(BSTNode<T> current, String codigo){
   //…
}

public ArrayList<T> decode(String mensaje){
   String[] codigo = mensaje.split(" ");
   ArrayList<T> result = new ArrayList<T>();
   for (int i=0; i<codigo.length; i++)
      result.add(decode(root, codigo[i]));
   return result;
}
```

Realizad un seguimiento exhaustivo (**actividad02.pdf**) del método code(), tomando el árbol que se muestra en la siguiente figura como estructura de datos de ejemplo y partiendo de la frase [40, 10, 50].



Ejercicio 8. Para comprobar el rendimiento de los árboles binarios de búsqueda vistos en clase (`BSTree<T>`, `AVLTree<T>` y `RBTree<T>`), se propone un test (*testPathHeightAllTress()*) que crea un `ArrayList<T>` con diferentes elementos (objetos de tipo `Item`) y, a continuación, baraja dicha colección (`shuffle`). Una vez barajada la muestra, se insertan dichos elementos en cada estructura ABB (obteniendo los diferentes tiempos de ejecución). Después, procesamos el `ArrayList<T>` y, para cada elemento del mismo, llamamos a la función `pathHeight()` de su respectivo ABB, manteniendo un contador de su profundidad respecto a la raíz. Finalmente, se calcula la profundidad media (*average*) de un elemento en cada uno de los tres ABBs diferentes. Como veréis, este test **no es usual** (ya resuelto y con instrucciones tipo `syso()`) y lo único que hace es mostrar en Consola diferentes datos acerca de tiempos de ejecución, la altura de cada estructura arborescente, así como la media de altura de cada uno de los elementos que los conforman. Como ejercicio (**actividad02.pdf**)

vamos a realizar lo que se indica a continuación:

- Vamos obtener y analizar distintos resultados modificando el tamaño de la muestra (10, 100, 1.000). Este proceso lo vamos a poder automatizar o, si preferís, vamos a hacer una ejecución distinta para cada tamaño de muestra. En la documentación tendréis que indicar, razonadamente, cuál ha sido vuestra decisión. A continuación, se comentarán los resultados obtenidos, basando vuestro análisis en una gráfica comparativa…de tiempos de ejecución y de altura media de cada estructura.
- ¿Qué pasaría si eliminamos la instrucción *Collection.shuffle(datos)*? ¿Cuál sería el efecto en los resultados obtenidos? Comentad la funcionalidad de dicha instrucción y repetid el proceso anterior, comentando detalladamente cuáles han sido los nuevos resultados obtenidos y por qué. Basad vuestro análisis en gráficas comparativas.