UNIDAD TEMÁTICA 4:

Arboles Binarios I

PRÁCTICOS DOMICILIARIOS INDIVIDUALES - FORMULACIÓN DE PSEUDOCÓDIGO

Escenario para todos los ejercicios: Estos ejercicios tratan del desarrollo de algoritmos en pseudocódigo y análisis del tiempo de ejecución correspondiente para el TDA ArbolBinarioBusqueda – TArbolBB- (y el correspondiente TDA ElemntoArbolBinariobBusqueda – TElementoABB).

Ejercicio #1

Operaciones Complementarias – pseudocódigo y análisis

1. Obtener la menor clave del árbol.

LENGUAJE NATURAL:
Al trabajar con un árbol binario de búsqueda, el nodo más a la izquierda será el menor, por lo que se recorre el árbol por la izquierda hasta encontrar el menor elemento.

PRECONDICIONES:
-El árbol debe de existir.

POSTCONDICIONES:
-El árbol original no se verá modificado.
-Retornará correctamente la menor clave.

PSEUDOCÓDIGO:

```
buscarMenorClave() :Comparable
COMIENZO
    SI (this.hijoIzq = nulo) ENTONCES
        DEVOLVER this.etiqueta()
    FIN SIN
    DEVOLVER this.hijoIzq.buscarMenorClave()
FIN
```

ORDEN DEL TIEMPO DE EJECUCIÓN:
Es O(log(n)), ya que no recorre completamente el árbol, sino que se llama recursivamente hasta llegar al nodo más a la izquierda, siendo que cada línea es de O(1).

2. Obtener la mayor clave del árbol.

LENGUAJE NATURAL:
Al trabajar con un árbol binario de búsqueda, el nodo más a la derecha será el mayor, por lo que se recorre el árbol por la derecha hasta encontrar el mayor elemento.

PRECONDICIONES:
-El árbol debe de existir.

POSTCONDICIONES:
-El árbol original no se verá modificado.
-Retornará correctamente la mayor clave.

PSEUDOCÓDIGO:
```
buscarMayor() : Comparable
COMIENZO
    SI (this.hijoDer = nulo) ENTONCES
        RETORNAR this.etiqueta()
    FIN SI
    RETORNAR this.hijoDer.buscarMayor()
FIN
```

ORDEN DEL TIEMPO DE EJECUCIÓN:
Es O(log(n)), ya que no recorre completamente el árbol, sino que se llama recursivamente hasta llegar al nodo más a la derecha, siendo que cada línea es de O(1).

3. Obtener la clave inmediata anterior a una clave dada (pasada por parámetro)

LENGUAJE NATURAL: Recibe por parámetro una clave y se devolverá a su predecessor. Con llamadas recursivas compara si la clave del nodo en donde se encuentra coincide con la ingresada por parámetro para devolver la clave inmediata anterior a la clave dada.

PRECONDICIONES:

-El árbol debe de existir.

POSTCONDICIONES:

-El árbol original no se verá modificado.

-Retornará correctamente la clave inmediatamente anterior.

PSEUDOCÓDIGO:

```
obtenerAnterior(Comparable etiqueta) : Comparable
COMIENZO
    RETORNAR obtenerAnterior(etiqueta, null)
FIN
obtenerAnterior(Comparable etiqueta, Comparable predecessor) : Comparable
COMIENZO
    SI (this.etiqueta.equals(etiqueta)) ENTONCES
        SI (this.hijoIzq <> nulo) ENTONCES
            RETORNAR this.hijoIzq.buscarMayor().getEtiqueta()
        FIN SI
    SINO
        SI (etiqueta.compareTo(this.etiqueta) < 0) ENTONCES
            SI (this.hijoIzq <> nulo) ENTONCES
                RETORNAR this.hijoIzq.obtenerAnterior(etiqueta, predecessor)
        SINO
            predecesor = this.etiqueta
            SI (this.hijoDer <> nulo) ENTONCES
                RETORNAR this.hijoDer.obtenerAnterior(etiqueta, predecesor)
            FIN SI
        FIN SI
    FIN SI
    RETORNAR predecesor
FIN
```

ORDEN DEL TIEMPO DE EJECUCIÓN:

En el peor de los casos será de O(n) si el nodo ingresado por parámetro es el último.

4. Obtener la cantidad de nodos de un nivel dado (por parámetro)

LENGUAJE NATURAL:
Recibe por parámetro un nivel y se devolverá la cantidad de nodos que hay en el mismo a través de llamadas recursivas que permitirá recorrer el árbol hasta llegar al nivel deseado.

PRECONDICIONES:
-El árbol debe existir.

POSTCONDICIONES:
-El árbol original no se verá modificado
-Retornará correctamente la cantidad de nodos del nivel dado

PSEUDOCÓDIGO

```
cantidadDeNodosPorNivel(int nivel) : int
COMIENZO
    SI (nivel = 0) ENTONCES
        RETORNAR 1
    FIN SI
    SI (hijoIzq <> nulo) ENTONCES
        RETORNAR hijoIzq.cantidadDeNodosPorNivel(nivel-1)
    FIN SI
    SI (hijoDer <> nulo) ENTONCES
        RETORNAR hijoDer.cantidadDeNodosPorNivel(nivel-1)
    FIN SI
    RETORNAR resultado
FIN
```

ORDEN DEL TIEMPO DE EJECUCIÓN:
En el peor de los casos será de O(n) si el nivel ingresado por parámetro es el último.

5. Listar todas las hojas cada una con su nivel.

LENGUAJE NATURAL:

Se recorre el árbol hasta llegar a sus hojas para devolver la etiqueta de las hojas y sus respectivos niveles.
Se hará a través de la recursión, en cada llamada recursiva se fija si el nodo actual es una hoja para devolverla junto a su nivel.

PRECONDICIONES:

-El árbol debe de existir.

POSTCONDICIONES:

-El árbol original no se verá modificado.
-Retornará correctamente las hojas y sus niveles.

PSEUDOCÓDIGO:

```
obtenerHojasConNivel(int cont) : String
COMIENZO
    SI (hijoIzq = nulo AND hijoDer = nulo) ENTONCES
        RETORNAR " " + this.etiqueta() + ", Nivel: "  + cont
    SINO SI (hijoIzq = nulo AND hijoDer <> nulo) ENTONCES
        RETORNAR  hijoDer.obtenerHojasConNivel(cont + 1)
    SINO SI (hijoIzq <> nulo AND hijoDer = nulo) ENTONCES
        RETORNAR  hijoIzq.obtenerHojasConNivel(cont + 1)
    SINO
        RETORNAR hijoDer.obtenerHojasConNivel(cont + 1) +
hijoIzq.obtenerHojasConNivel(cont + 1)
    FIN SI
FIN
```

ORDEN DEL TIEMPO DE EJECUCIÓN:

O(n) ya que necesitará recorrer todo el árbol para llegar a sus hojas.

6. Verificar si el árbol es de búsqueda.

LENGUAJE NATURAL:

Dado un árbol, devolverá si el mismo es de búsqueda o no.
Esto lo hará chequeando que los hijos (si existen) de un nodo serán: el izquierdo

más pequeño que el nodo y el derecho más grande.

PRECONDICIONES:

-El árbol debe de existir

POSTCONDICIONES:

-El árbol no se verá modificado

-Retornará correctamente si es un árbol de búsqueda

PSEUDOCÓDIGO:

```
esABB() : booleano
COMIENZO
	SI (hijoIzq = nulo AND hijoDer = nulo) ENTONCES
		RETORNAR true
	SINO SI (hijoIzq = nulo AND hijoDer <> nulo) ENTONCES
		RETORNAR (this.getEtiqueta().compareTo(hijoDer.getEtiqueta()) < 0) =
hijoDer.esABB();
	SINO SI (hijoIzq <> nulo AND hijoDer = nulo) ENTONCES
		RETORNAR (this.getEtiqueta().compareTo(hijoDer.getEtiqueta()) > 0) =
hijoDer.esABB();
	SINO
		RETORNAR ((this.getEtiqueta().compareTo(hijoDer.getEtiqueta()) < 0) =
hijoDer.esABB()) =  ((this.getEtiqueta().compareTo(hijoDer.getEtiqueta()) > 0) =
hijoDer.esABB())
	FIN SI
FIN
```

ORDEN DEL TIEMPO DE EJECUCIÓN:

O(n) ya que debe recorrer todo el árbol para verificar que efectivamente es un árbol binario.

Ejercicio #2

Implementar en JAVA, dentro de los TDA señalados, las operaciones indicadas.

1. Obtener la menor clave del árbol.

```
class TElementoAB<T> implements IElementoAB<T> {
    /**
     * Devuelve el menor elemento.
     *
     * @return TElementoAB<T>.
     */
    @Override
    public TElementoAB<T> obtenerMenorElemento() {
        if (this.getHijoIzq() == null) {
            return this;
        }
        return this.getHijoIzq().obtenerMenorElemento();
    }
```

2. Obtener la mayor clave del árbol.

```
class TElementoAB<T> implements IElementoAB<T> {
    /**
     * Devuelve el mayor elemento.
     *
     * @return TElementoAB<T>.
     */
    @Override
    public TElementoAB<T> obtenerMayorElemento() {
        if (this.getHijoDer() == null) {
            return this;
        }
        return this.getHijoDer().obtenerMayorElemento();
    }
```

3. Obtener la clave inmediata anterior a una clave dada (pasada por parámetro)

```
public class TArbolBB<T> implements IArbolBB<T> {
    @Override
    public Comparable obtenerClaveInmediataAnterior(Comparable etiqueta) {
        return raiz.obtenerClaveInmediataAnterior(etiqueta);
    }
    @Override
    public Comparable obtenerClaveInmediataAnterior(Comparable etiqueta,
Comparable predecesor) {
        return raiz.obtenerClaveInmediataAnterior(etiqueta, predecesor);
    }
```

```
class TElementoAB<T> implements IElementoAB<T> {
    @Override
    public Comparable obtenerClaveInmediataAnterior(Comparable etiqueta) {
        return obtenerClaveInmediataAnterior(etiqueta, null);
    }
    @Override
    public Comparable obtenerClaveInmediataAnterior(Comparable etiqueta,
Comparable predecesor) {
        if (this.etiqueta.equals(etiqueta)) {
            if (this.hijoIzq != null) {
                return this.hijoIzq.obtenerMayorElemento().getEtiqueta();
            }
        }
        else if (etiqueta.compareTo(this.etiqueta) < 0) {
            if (this.hijoIzq != null) {
                return this.hijoIzq.obtenerClaveInmediataAnterior(etiqueta,
predecesor);
            }
        }
        else {
            predecesor = this.etiqueta;
            if (this.hijoDer != null) {
                return this.hijoDer.obtenerClaveInmediataAnterior(etiqueta,
predecesor);
            }
        }
        return predecesor;
    }
```

4. Obtener la cantidad de nodos de un nivel dado (por parámetro)

```
public class TArbolBB<T> implements IArbolBB<T> {
     public int cantidadDeNodosPorNivel(int nivel){
          if (esVacio()) {
               return 0;
          }     else {
               return raiz.cantidadDeNodosPorNivel(nivel);
          }
     }
```

```
class TElementoAB<T> implements IElementoAB<T> {
     public int cantidadDeNodosPorNivel(int nivel){
          if(nivel == 0){
               return 1;
          }
          else if(hijoIzq != null){
               return hijoIzq.cantidadDeNodosPorNivel(nivel-1);
          }
          else{
               return hijoDer.cantidadDeNodosPorNivel(nivel-1);
          }

     }
```

5. Listar todas las hojas cada una con su nivel.

```
public class TArbolBB<T> implements IArbolBB<T> {
    @Override
    public String obtenerHojasConNivel(int cont) {
        if (esVacio()) {
            return "No hay hojas";
        } else {
            return raiz.obtenerHojasConNivel(cont);
        }
    }
```

```
class TElementoAB<T> implements IElementoAB<T> {
    @Override
    public String obtenerHojasConNivel(int cont) {
        if (hijoIzq == null && hijoDer == null) {
            return "(Hoja: " + this.etiqueta + ", Nivel: " +
cont + ") ";
        } else if (hijoIzq == null && hijoDer != null) {
            return hijoDer.obtenerHojasConNivel(cont + 1);
        } else if (hijoIzq != null && hijoDer == null) {
            return hijoIzq.obtenerHojasConNivel(cont + 1);
        } else {
            return hijoDer.obtenerHojasConNivel(cont + 1) +
hijoIzq.obtenerHojasConNivel(cont + 1);
        }
    }
```

6. Verificar si el árbol es de búsqueda.

```
public class TArbolBB<T> implements IArbolBB<T> {
    @Override
    public boolean esABB() {
        if (esVacio()) {
            return true;
        } else {
            return raiz.esABB();
        }
    }
```

```
class TElementoAB<T> implements IElementoAB<T> {
    @Override
    public boolean esABB() {
        if (hijoIzq == null && hijoDer == null) {
            return true;
        } else if (hijoIzq == null && hijoDer != null) {
            return
(this.getEtiqueta().compareTo(hijoDer.getEtiqueta()) < 0) ==
hijoDer.esABB();
        } else if (hijoIzq != null && hijoDer == null) {
            return
(this.getEtiqueta().compareTo(hijoIzq.getEtiqueta()) > 0) ==
hijoIzq.esABB();
        } else {
            return
((this.getEtiqueta().compareTo(hijoDer.getEtiqueta()) < 0) ==
hijoDer
                    .esABB()) ==
((this.getEtiqueta().compareTo(hijoIzq.getEtiqueta()) > 0) ==
hijoIzq.esABB());
        }
    }
```