Análisis de Complejidad

Estructura de Datos

UNIVERSIDAD DE BURGOS

José Luis Garrido Labrador

José Miguel Ramírez Sanz

Índice de contenido

1. Método altura

```
public int altura(E elemento) {
	List<Nodo> lista = buscar(super.raiz, elemento);
	Nodo supp = lista.get(0);
	if (supp == null) {
		return -1;
	}
	return alturaR(supp);
}
```

Teniendo en cuenta que el método llama a otro método recursivo (alturaR) del que no podemos saber con certeza su complejidad para sumar a la complejidad de este hemos decidido hacer una estimación "a ojo". Para comenzar con el cálculo sacamos la complejidad del método buscar del ArbolBB que basándonos en los apuntes sabemos que es O(lg n). El método alturaR creemos que por su estructura tiene una complejidad de $O(lg^2\ n)$ ya que su funcionamiento es similar al peor caso de buscar de ArbolBB, ya que lo que hace es recorrer los distintos subárboles hasta sus hojas más profundas. Por lo tanto, la complejidad basándonos puramente en la especulación sería de $O((lg^2\ n)^2)$.

2. Método alturaR

```
private int alturaR(Nodo nodo) {
	int alturaD = 0;
	int alturaI = 0;
	if (nodo.getDer() == null && nodo.getIzq() == null) {
		return 0;
	}
	if (nodo.getDer() == null) {
		alturaI = 1 + alturaR(nodo.getIzq());
	} else if (nodo.getIzq() == null) {
		alturaD = 1 + alturaR(nodo.getDer());
	} else {
		alturaI = 1 + alturaR(nodo.getIzq());
		alturaD = 1 + alturaR(nodo.getDer());
	}
	if (alturaD > alturaI) {
		return alturaD;
	}
	return alturaI;
}
```

Como ya hemos comentado, este método es recursivo y en el peor de los casos el nodo inicial será la raíz. La función se llama así misma en dos ocasiones como máximo, pero sin anidar, por lo que

aplicando la regla de la suma nos quedamos con la complejidad de la propia función. Por tanto manteniendo en el método anterior deducimos que la complejidad es aproximadamente $O(lg^2 n)$.

3. Método profundidad

```
public int profundidad(E elemento) {
	List<Nodo> lista = buscar(super.raiz, elemento);
	Nodo supp = lista.get(0);
	if (supp == null) {
		return -1;
	}
	if (lista.get(1) == null) {
		return 0;
	}
	return 1 + profundidad(lista.get(1).getDato());
}
```

Como en los casos anteriores, este método también es recursivo, para hacer su análisis hemos conocido primero la complejidad del método buscar de ArbolBB que es O(lg n), luego profundidad se vuelve a llamar a si mismo hasta llegar a la raíz, y en el peor de los casos la profundidad es igual a la altura del árbol, que para simplificar consideraremos que en este caso es h. Por tanto, la complejidad de este método sería O(lg n * h).

4. Método comprobarAVL

```
private int comprobarAVL(Nodo nodo) {
	int alturaD;
	int alturaI;
	if (nodo == null) {
		return 0;
	}

	if (nodo.getDer() == null) {
		alturaD = 0;
	} else {
		alturaD = 1 + altura(nodo.getDer().getDato());
	}
	if (nodo.getIzq() == null) {
		alturaI = 0;
	} else {
		alturaI = 1 + altura(nodo.getIzq().getDato());
	}
	return alturaD - alturaI;
```

```
}
```

Este método por si mismo tiene una complejidad de O(1) si los métodos altura no tuviesen la complejidad que tienen, por tanto teniendo en cuenta que aplicamos la regla de la suma, la complejidad del método comprobarAVL será la misma que el de altura, $O((\lg^2 n)^2)$.

5. Método encontrarDes

```
private Nodo encontrarDes(Nodo nodo) {
		int factor = comprobarAVL(nodo);
		if(factor == 2 || factor == -2)
			return nodo;
		else if(nodo==raiz)
			return null;
		else
			return encontrarDes(buscar(raiz,nodo.getDato()).get(1));
	}
```

Este método utiliza comprobarAVL que tiene una complejidad de $O((\lg^2 n)^2)$ y de buscar que tiene una complejidad de $O(\lg^2 n)$ y aplicando la regla de la suma nos quedamos que la complejidad de este método es de $O(((\lg^2 n)^2)^2)$ porque el método también es recursivo.

6. Métodos rotacionIzquierda y rotacionDerecha

```
	private void rotacionIzquierda(Nodo nodo) {
		Nodo raizLocal;
		raizLocal = nodo.getDer();
		nodo.setDer(raizLocal.getIzq());
		raizLocal.setIzq(nodo);
		Nodo padre = buscar(super.raiz, nodo.getDato()).get(1);
		if (padre == null)
			super.raiz = raizLocal;
		else if (padre.getDer() == nodo)
			padre.setDer(raizLocal);
		else
			padre.setIzq(raizLocal);
	}
```

Estos dos métodos, prácticamente iguales, hacen uso de la función buscar que tiene una complejidad de O(lg n), a parte de esto solo hacen uso de sentencias de complejidad O(1), por tanto la complejidad real de ambos métodos es de O(lg n) aplicando la regla de la suma.

7. Métodos rotacionDerechaIzquierda y rotacionIzquierdaDerecha

```
private void rotacionIzquierdaDerecha(Nodo nodo) {
          rotacionIzquierda(nodo.getIzq());
          rotacionDerecha(nodo);
     }
```

Aplicando la regla de la suma sobre estos dos métodos apartir de las complejidades de los métodos mencionados justo en el punto anterior (O(lg n) en cada caso) estos métodos también tienen la misma complejidad, O(lg n).

8. Método requilibrioAVL

```
private void reequilibrioAVL(Nodo nodo) {
          if (nodo != null) {
               int factor, derecho, izquierdo;
               factor = this.comprobarAVL(nodo);
               derecho = this.comprobarAVL(nodo.getDer());
               izquierdo = this.comprobarAVL(nodo.getIzq());
               switch (factor) {
               case 2:
                    switch (derecho) {
                    case 1:
                    case 0:
                         rotacionIzquierda(nodo);
                         break;
                    case -1:
                         rotacionDerechaIzquierda(nodo);
                         break;
                    }
                    break;
               case -2:
                    switch (izquierdo) {
                    case -1:
                    case 0:
                         rotacionDerecha(nodo);
                         break;
                    case 1:
                         rotacionIzquierdaDerecha(nodo);
                         break;
                    }
                    break;
```

```
                }
        }
}
```

Aplicando la regla de la suma, el método que es llamado desde requilibrioAVL que tiene mayor complejidad (estimada) es comprobarAVL que tiene una complejidad de $O((\lg^2 n)^2)$.

9. Método add

```
@Override
public boolean add(E elemento) {
        if (!super.add(elemento)) {
                return false;
        }
        Nodo padre = super.buscar(raiz, elemento).get(1);
        if(padre!=null)
                reequilibrioAVL(encontrarDes(padre));
        return true;
}
```

Este método hace uso de otros dos métodos recursivos y el método add del padre (super.add). Add tiene una complejidad de O(lg n), el método buscar tiene una complejidad idéntica, O(lg n), y por último los métodos requilibrioAVL y encontrarDes tiene complejidad $O((\lg^2 n)^2)$ y $O(((\lg^2 n)^2)^2)$ respectivamente. Teniendo en cuenta esta complejidades y aplicando la regla de la suma nos quedamos con una complejidad de $O(((\lg^2 n)^2)^2)$.

10. Método remove

```
@Override
public boolean remove(Object o){
        Nodo padre;
        try{
                padre = super.buscar(raiz, (E) o).get(1);
        }catch(ClassCastException ex){
                return false;
        }
        boolean retorno = super.remove(o);
        if(retorno){
                if(padre!=null)
                        reequilibrioAVL(encontrarDes(padre));
                else{
                        reequilibrioAVL(encontrarDes(raiz));
```

```
                }
            }
            return retorno;
        }
```

El método remove tiene la misma complejidad que el método add porque la única diferencia en la suma de complejidades es la llamada al método remove del padre (super.remove) que tiene la misma complejidad que el método super.add ya que es un árbol de búsqueda, por tanto como la suma se mantiene el resultado es el mismo, $O(((\lg^2 n)^2)^2)$.

11. Método tratar

```
private void tratar(Stack<Integer> veces) {
            Integer elemento = veces.pop();
            elemento++;
            veces.push(elemento);
        }
```

Como se puede observar hay dos llamadas a funciones externas, de la clase Stack, que al ser inserción y extracción de una pila tiene complejidad de O(1).

12. Métodos de recorridos

```
    public List<E> preOrden() {
        List<E> retorno = new ArrayList<>(this.size());
        Stack<Integer> pila = new Stack<>();
        Stack<Nodo> pilaNodos = new Stack<>();
        pilaNodos.push(super.raiz);
        pila.push(1);
        while (pilaNodos.peek() != super.raiz || pila.peek() != 3) {
            Nodo cabeza = pilaNodos.peek();
            Integer veces = pila.peek();
            if (veces == 3) {
                pila.pop();
                pilaNodos.pop();
            } else if (veces == 1) {
                tratar(pila);
                retorno.add(cabeza.getDato());
                cabeza = cabeza.getIzq();
                if (cabeza != null) {
                    pila.push(1);
```

```
                        pilaNodos.push(cabeza);
                    }
                } else {
                    tratar(pila);
                    cabeza = cabeza.getDer();
                    if (cabeza != null) {
                        pila.push(1);
                        pilaNodos.push(cabeza);
                    }
                }
            }

            return retorno;
        }
```

En estos tres métodos se hacen llamadas a métodos externos de complejidad O(1) y existe un bucle while que recorre cada elemento tres veces, por lo que la complejidad es de O(3n) que simplificando es O(n).