2020

UNIVERSIDAD DE BURGOS

ESTRUCTURA DE DATOS

ALEJANDRO ORTEGA;JORGE RUIZ

Contenido

Introducción de la practica

La cuarta práctica entregable consiste en la modificación de los distintos métodos de un árbol de búsqueda binaria (ABB) para convertirlo en un árbol binario equilibrado (AVL)

Utilizaremos rotaciones a la hora de añadir y borrar elementos, e implementaremos los tres tipos de recorridos (**inOrder, posOrder y preOrder)**

Complejidad Algorítmica

En este apartado veremos la complejidad algorítmica de cada uno de los métodos.

Hemos aplicado la regla del producto para los métodos que son llamados de forma recursiva.

ArbolAVL()

Es el constructor de la clase, y llama al constructor de su clase padre **ArbolBB.**

Dicho constructor está vacío, por lo que suponemos que la clase tiene una complejidad de **O(1).**

reequilibrioAVL(Nodo)

Llama varias veces consecutivas a la función calcFactorDesequilibrio() y realiza rotaciones. Esta llama se realiza **log(n)** veces, por lo que la complejidad de este método aplicando la regla del producto es de log(n)^2 = **O(log(n)).**

add(E)

Utiliza el método **super.add(E),** y dicho método utiliza la búsqueda binaria.

La búsqueda binaria tiene una complejidad media de O(log(n)).

Nuestra implementación utiliza el método **reequilibrioAVL(Nodo),** el cual es llamado **log(n)** veces.

La complejidad final es de O(log(n)) + O(log(n)), que aplicando la regla de la suma se mantiene en **O(log(n))**

remove(E)

Llama a la función **super.remove(E)** y además a la función **reequilibrioAVL** log(n) veces.

La complejidad del equilibrio es la misma que en el método add(E) O(log(n))

super.remove(E) llama varias veces consecutivas al método **buscar()**, ninguna es en un bucle, por lo que la complejidad final del método **super.remove(E)** aplicando la regla de la suma es **log(n).**

Si aplicamos la regla de la suma con los dos métodos de nuestro remove es de **O(log(n)).**

profundidad(E)

Con un bucle while recorremos el árbol hasta encontrar un dato. Podríamos decir que este método es un método **buscar(E)** con un contador que calcula la profundidad del nodo. Tiene una complejidad **mejor de O(1)** si el elemento a encontrar el primero**,** y una complejidad peor y promedio de **O(log(n))**.

El método buscar implementado tiene una complejidad peor de O(n) (si todos los nodos cuelgan de la misma rama), pero **profundidad()** se ejecuta con el árbolAVL, por lo que al tener el árbol siempre equilibrado, la complejidad sólo puede ser O(n) cuando el número de elementos es 2.

altura(E) y altura(Nodo)

Utiliza la búsqueda de un dato O(log(n)) y se llama de forma recursiva (log(n) veces) si son llamadas desde desde la raíz.

La complejidad algorítmica de esta función es log(n^2) = 2*log(n) = **O(log(n)).**

alturaDer(Nodo) y alturaIzq(Nodo)

Llaman a la función de la altura de un nodo (izquierdo o derecho). La complejidad algorítmica de esta función llamada desde la raíz es de **O(log(n)).**

calcFactorDesequilibrio()

Llama dos veces a la función **alturaDer(Nodo)** y **alturaIzq(Nodo)** con el Nodo raíz.

La complejidad algorítmica de este método es de log(n) + log(n) que aplicando la regla de la suma, la complejidad algorítmica de este método es de **O(log(n))**

Rotaciones

Agrupamos los 4 métodos de rotaciones (rotacionDerecha(), rotacionIzquierda(), rotacionIzqDer() y rotacionDerIzq()).

Los 4 métodos únicamente mueven punteros y hacen operaciones básicas, por lo que la complejidad algorítmica de estos métodos es de **O(1)**

nodoAnterior(nodo)

Llama a la función buscar, que devuelve un nodo y su nodo padre, la complejidad promedia de este método es de **O(log(n)).**

Recorridos

Los tres tienen una complejidad algorítmica de O(n), y que se accede mínimo 1 vez a cada nodo, y en algunos casos, se acceden varias veces a cada nodo. Se utilizan estructuras auxiliares donde los métodos utilizados tienen complejidades simples **O(1).**

inOrden()

Encontramos dos pilas (una que almacena los datos, y otra el recorrido), y un bucle while con el que recorremos todo el árbol.

El coste de recorrer todo el árbol es de O(n), y los accesos a la pila tienen un coste de O(1). Como resultado, la complejidad de este método es de O(n).

preOrden()

En este método también encontramos 2 pilas y un bucle while. Recorre todo el árbol por lo que tiene una complejidad de O(n).

posOrden()

Este método, al igual que los otros dos, utiliza dos pilas y un bucle final para recorrer el árbol. El recorrido del árbol tiene una complejidad de O(n), pero esta vez nos apoyamos de dos hashmaps que nos ayudan a llevar un registro de nodos visitados.

Los métodos utilizados son .put() y .containsKey(), que tienen una complejidad de **O(1).**

Si bien este recorrido tiene una lógica un poco más compleja, sigue teniendo una complejidad de **O(n).**