Instituto Tecnológico y de Estudios Superiores de Monterrey.

PROGRAMACIÓN DE ESTRUCTURA DE DATOS Y ALGORITMOS FUNDAMENTALES.

Actividad 3.4: Actividad Integradora .

Profesor: *Luis Humberto González Guerra.*

Alumna*:* ***A00831510*** Andrea Marisol Salcedo Vázquez

Grupo 1

24/10/2021

Reflexión personal

Andrea Marisol Salcedo Vázquez

Para esta actividad usamos los archivos que ordenamos anteriormente con las LinkedList para el mar mediterráneo y el mar rojo. Como se debía realizar lo mismo para ambos archivos decidimos crear una función y de esta forma no repetir código. En esta función se recibe el nombre del archivo que se va a leer y el árbol que vamos a modificar. Como la función recibe el árbol y no una copia de este entonces se va a modificar y por eso mismo regresa nada nuestra función.

Creamos un struct llamado dataCs que nos ayudará a almacenar los datos de cada entrada conteniendo el ubi y la cantidad de entradas que tuvo. Le pusimos un constructor con parámetros para que al momento de crearlo le mandáramos el ubi y la cantidad y no tener que usar setters, y de igual forma se creó el constructor sin parámetros. Además, se hizo la sobrecarga del operador == el cual compara la cantidad y el ubi verificando que sean iguales. La sobrecarga del operador > el cual verifica primero la cantidad y en el caso que sean iguales checa cual ubi es mayor, si son diferentes checa que cantidad es mayor y la sobrecarga del operador = para poder igualar una dataCs a otra. Estas sobrecargas las hicimos debido a que nuestro árbol va a ser de nodos de tipo dataCs y en el momento de agregar un nodo debe compararlos y verificar si algún nodo es mayor, menor o igual para saber en qué lugar agregarlo. De igual manera se hizo la sobrecarga del operador << haciendo esta función friend para que pueda ser usada fuera del struct.

Lo primero que realiza es abrir el archivo e inicializar un contador en 1. Lo inicia en uno porque dentro del while ponemos que nuestro contador aumente después de que verifique si cumple con una condición. En este while es en el que vamos a contar la cantidad de series y las vamos a añadir al árbol. Por lo que primero fuera del árbol creamos una variable llamada serieAnt que nos va a ayudar a checar en qué momento la serie cambia. Una vez dentro del while vamos obteniendo los ubis de cada línea y con el if checamos si cambia o si sigue siendo la misma. En el caso de que cambie entonces creamos un nuevo dato llamado dataCs el cual es un struct que creamos para almacenar los datos, y añadimos este dato a nuestro árbol. Finalmente volvemos el contador 0 y cambiamos la serieAnt por la nueva serie que acaba de aparecer. El contador se inicializa en 0 porque se va a incrementar en uno una vez que salga del if.

Una vez que ya haya terminado de leer el archivo agrega la ultima serie encontrada ya que esta no se añade durante el while y cierra el archivo. Esta función nos quedó de complejidad O(n*height) esto es debido a que en el archivo lee cada dato solo una vez, volviéndolo n, pero cada que entre al if va a llamar a nuestra función add del árbol la cual tiene una complejidad de O(height), y en el peor de los casos todas nuestras ubis serían diferentes por lo que entrará n veces al height volviéndolo una complejidad de O(n*height).

Como nos pudimos dar cuenta en los párrafos anteriores para esta actividad hicimos uso de la estructura de árboles binarios, esto nos ayudó a guardar los datos por la cantidad de entradas que tuvieron y si había empate por ubi. En un futuro si se quisiera buscar un buque

con cierta cantidad de entradas sería más fácil pues solo debe ir comparando nodo por nodo y viendo si la cantidad que se muestra es mayor o menor e ir a la izquierda o a la derecha según sea el caso por lo que iría descartando todo un lado del árbol, volviendo la complejidad de la búsqueda O(height) dependiendo esto de a qué altura se encuentre el nodo que estamos buscando.

De igual forma, si queremos agregar un dato tiene una menor complejidad que con las LinkedList ya que si bien si queremos agregar un dato a una linkedlist al inicio este es O(1) cuando lo queremos agregar al final nos da una complejidad O(n) ya que debe pasar por todos los nodos. Además de que ordenar una LinkedList es una complejidad de O(n^2) ya que lo realiza secuencialmente. Pero con los árboles de búsqueda binaria se va ordenando automáticamente facilitando de esta forma el trabajo para imprimirlo ordenadamente.

Para imprimir los datos guardados en el árbol de manera descendente utilizamos el método de inorderConvexo, esto es debido a que el inorder nos lo imprime de manera ascendente, por lo que para imprimirlo de manera descendente cambiamos a que nos imprimiera primero los de la derecha y luego los de la izquierda ya que de esta forma imprime primero los mayores y finalmente los menores. Teniendo una complejidad de O(n) puesto que solo pasa por un nodo cada vez.

Como podemos ver el usar árboles de búsqueda binaria tiene muchos beneficios ya que se va ordenando automáticamente mientras añadimos un dato. Sin embargo, puede existir el caso en el que le den los datos ordenados de manera ascendente o descendente y todos los nodos queden a la izquierda o a la derecha haciendo esto como si fuera una LinkedList, pero en los otros casos, especialmente en los que queda balanceado logramos que cuando busquemos un dato se vaya descartando una gran parte de los nodos haciendo esto más fácil y rápido.