Laboratorio N°4
Estructura de datos y algoritmos

Integrantes: Hugo Rojas e Isidora González
Profesor laboratorio: Mauricio Hidalgo
Profesor cátedra: Yerko Ortiz
Sección 04
Mails: hugo.rojas1@mail.udp.cl isidora.gonzalez4@mail.udp.cl

Introducción

En el siguiente informe se dará a conocer la modificación y mejora de un código que fue visto previamente, del cual es posible recordar que una empresa hizo entrega de un código el cual traía incluído clases y métodos, donde algunos de ellos fueron eliminados, modificados u ambos, con el objetivo de poder trabajar con una plataforma de videos musicales. Además, fueron implementados métodos para la búsqueda de videos por título, una función para poder revertir la lista entregada y por último, una función dirigida a la popularidad de los videos donde se retornaba el vídeo con más likes.

La empresa ahora está en busca de reestructurar la plataforma, ya que un trabajador renunció y dejó el código a medias, por lo que se hará la depuración de ciertos métodos incorrectos para así poder hacer la implementación de un buscador de videos según el nombre de los canales y para esto se debe hacer una agrupación de videos por canal con el nombre **“ChannelTitle”**, luego de esto las listas de cada canal se ordenan en un árbol de AVL. Además se hará la implementación de los métodos creados previamente en el archivo **“Cliente”** y corroborar el correcto funcionamiento junto con los códigos nuevos.

Se inició la corrección del código depurando el constructor creado en la clase **“AVLTree”** y de los métodos **“Insert”** que inserta un canal en el AVL, **“Balance”** que se encarga de actualizar las alturas de los árboles y verifica si está desbalanceado o no para realizar las rotaciones respectivas, **“rotateLeft”** y **“rotateRight”** que realizan rotaciones a la derecha e izquierda respectivamente.

Luego de eso es posible hacer la implementación de los métodos **“Delete”** el cual recibe como llave el título del canal y lo elimina del árbol sin retornar nada, y **“Find”** recibe como llave el título del canal y retorna la lista enlazada de vídeos correspondiente al mismo.
A continuación se hará un desglose de lo que fue crear y depurar cada método y función para su mejor comprensión.

Depuración

Avl tree:

Método Insert:

En este método se hizo un cambio en la primera condición de “if”, en donde se cambia **“node.right”** a **“node.left”** basándose en la lógica de orden de un árbol, ya que si se quieren insertar de manera correcta los nuevos nodos debe tener un orden establecido.

El siguiente cambio se hizo en el **“return”** del método, de **“return node”** por **“return balance(node)”**, ya que cada vez que se agrega un nuevo nodo el árbol queda desbalanceado, por lo que al llamar al “balance” en el return, este acomoda el árbol.

```
Node insertNode(Node node, String key, LinkedList videos){
    if(node == null){
        return new Node(key, videos);
    }
    if(key.compareTo(node.key) < 0)
        node.right = insertNode(node.left, key, videos);
    else if (key.compareTo(node.key) > 0) {
        node.right = insertNode(node.right,key, videos);
    }
    else {
        throw new RuntimeException("Duplicate key!");
    }
    return node;
}
```

Ilustración 1: (Previo a modificación)

```
Node insertNode(Node node, String key, LinkedList videos){
    if(node == null){
        return new Node(key, videos);
    }
    if(key.compareTo(node.key) < 0)
        node.left = insertNode(node.left, key, videos);
    else if (key.compareTo(node.key) > 0) {
        node.right = insertNode(node.right,key, videos);
    }
    else {
        throw new RuntimeException("Duplicate key!");
    }
    return balance(node);
}
```

Ilustración 2: (Después de la modificación)

Método Balance: En este método se realizó el primer cambio en el segundo **“if”** en donde se está igualando **“node.left = rotateLeft(node.left)”**, al cual se le asignó el valor **“rotateLeft(node.left)”** para el correcto balanceo del árbol, así el cambio que este sufre en la rotación pueda ser guardado.

El segundo cambio se puede visualizar en el **“return”** del primer **“if”**, el cual fue implementado para poder imprimir el nodo que fue rotado.

De igual manera que con el **“node.left = rotateLeft(node.left)”,** se crea lo mismo, pero con todas las variables de **“right”**.

```
Node balance(Node node){
    updateHeight(node);
    if (getBalance(node) < -1){
        if(height(node.left.left) < height(node.left.right)){
            //System.out.printf(node.left.key + " is rotated left and ");
            rotateLeft(node.left);

        }
        //System.out.println(node.key + " is rotated right" );
        rotateRight(node);
    }
    else if(getBalance(node) > 1){
        if(height(node.right.right) < height(node.right.left)){
            //System.out.print(node.right.key + " is rotated right and ");
            rotateRight(node.right);
        }
        //System.out.println(node.key +" is rotated left");
        rotateLeft(node);
    }
    return node;
}
```

Ilustración 3: (Previo a modificación)

```
Node balance(Node node) {
    updateHeight(node);
    if (getBalance(node) < -1) {
        if (height(node.left.right) > height(node.left.left)) {
            node.left = rotateLeft(node.left);
        }
        return rotateRight(node);
    } else if (getBalance(node) > 1) {
        if (height(node.right.left) > height(node.right.right)) {
            node.right = rotateRight(node.right);
        }
        return rotateLeft(node);
    }
    return node;
}
```

Ilustración 4: (Después de la modificación)

Método RotateRight:

En este método se crea la variable **“Node n3”** que guarda el nodo derecho de la variable **“Node n2”** y a diferencia del código anterior a esta función, se utiliza **“n2.right”** en vez de **“n1.right”** para una correcta ejecución del código.

Por otro lado, se tiene que **“n2.right = n1”** debido a que si se mantiene la igualación de la variable a **“n1.right”** como en el código anterior, se perdería la estructura del árbol en la rotación. A consecuencia de este cambio, se modifica **“n1.left = n3.left”** por **“n1.left = n3.”**

Por último, se modifica el **“return”** para que nos devuelva el nodo **“n2”** que pasaría a ser la raíz del árbol.

```
Node rotateRight(Node n1) {
    Node n2 = n1.left;
    Node n3 = n1.right;
    n2.right = n1;
    n1.left = n3.right;
    updateHeight(n1);
    updateHeight(n2);
    return n1;
}
```

Ilustración 5: (Previo a modificación)

```
Node rotateRight(Node n1) {
    Node n2 = n1.left;
    Node n3 = n2.right;
    n2.right = n1;
    n1.left = n3;
    updateHeight(n1);
    updateHeight(n2);
    return n2;
}
```

Ilustración 6: (Después de la modificación)

Método RotateLeft:

De igual manera que en el método de **“RotateRight”** se hicieron prácticamente los mismos cambios, ya que éste método también requería el cambio de igualación de su variable y el resto de ellas por consecuencia.

```
Node rotateLeft(Node n1) {
    Node n2 = n1.right;
    Node n3 = n1.left;
    n2.left = n1;
    n1.right = n3.left;
    updateHeight(n1);
    updateHeight(n2);
    return n1;
}
```

Ilustración 7: (Previo a modificación)

```
Node rotateLeft(Node n1) {
    Node n2 = n1.right;
    Node n3 = n2.left;
    n2.left = n1;
    n1.right = n3;
    updateHeight(n1);
    updateHeight(n2);
    return n2;
}
```

Ilustración 8: (Después de la modificación)

Implementación

Método Find: En este método se busca analizar todo el árbol buscando el canal deseado desde la consola, para esto se implementará:

```
Node find(Node node, String key) {
    if (node == null) {
        return null;
    }
    if (node.key.equals(key)) {
        return node;
    }
    Node encontradoIzq = find(node.left, key);
    if (encontradoIzq != null) {
        return encontradoIzq;
    }
    return find(node.right, key);
}
```

Ilustración 9: (Función find)

El primer **“if”** retornará **“null”** si es que no encuentra el nodo, el segundo **“if”** retorna el nodo que coincida con el nombre de éste y con el parámetro recibido, las líneas restantes recorren el árbol de manera recursiva.

Método Delete:

Este método busca eliminar el nodo en el que su nombre coincida con el parámetro recibido desde la consola.

```
private Node delete(Node root, String key) {
    if (root == null) return null;

    int aux = key.compareTo(root.key);
    if (aux < 0) {
        root.left = delete(root.left, key);
    } else if (aux > 0) {
        root.right = delete(root.right, key);
    } else {
        if (root.right == null) return root.left;
        if (root.left == null) return root.right;

        Node temp = root;
        root = GetMin(temp.right);
        root.right = deleteMin(temp.right);
        root.left = temp.left;
        root.height = height(root.left) + height(root.right) + 1;
    }
    return balance(root);
}
```

Ilustración 10: (Función delete)

Como primera instancia se tendrá en cuenta el caso en que no se encuentre el nodo, si no se encuentra, este retorna **“null”**. Con esto ahora se procederá a realizar la eliminación del nodo que coincida con el nombre, para luego recorrer recursivamente todo el árbol.

```
private Node deleteMin(Node root) {
    if (root.left == null) return root.right;
    root.left = deleteMin(root.left);
    root.height = height(root.left) + height(root.right) + 1;

    // Balancear el árbol después de la eliminación
    root = balance(root);
    return root;
}
Node GetMin(Node node){
    if(node == null){
        return null;
    }
    if(node.left != null){
        return GetMin(node.left);
    }
    return node;
}
```

Ilustración 11: (Funciones que se llaman en la función delete)

Estas dos funciones que se llaman dentro de la función **“Delete”** son las que permiten que el árbol se mantenga ordenado y equilibrado para finalmente retornar el nodo eliminado, pero dentro de la función **“balance”**, la cual permitirá balancear el árbol antes de imprimirlo.

Client:

En esta parte del código se realizará la implementación de la función **“delete”** y **“find”** dentro del **“Client”**, el cual permite mostrar el menú al usuario.
En concreto se implementará esto en el caso 2 y 3 del menú principal.

Case 2(Find):

```
case "2" -> {
    System.out.println("Ingrese nombre del canal a buscar");

    String targetChannel = br.readLine();
    AVLTree.Node target = tree.find(targetChannel);
    if(target != null){
        System.out.println("Canal encontrado");
        channelVideosMenu(target.videos, br);
    } else {
        System.out.println("Canal no encontrado");
    }

}
```

Ilustración 12: (Case 2)

Se crea la variable tipo string de nombre **“targetChannel”**, la cual permitirá preguntar al usuario qué nombre desea buscar, para luego llamar a la clase **“AVLTree”** y crear una variable **“target”**, la cual tendrá como valor el llamado de la función **“find”** de la clase **“AVL”,** teniendo como parámetro el video recibido desde

el main; Así luego se ingresa a un par de condiciones, las cuales mostrarán por pantalla si se encontró el video o no.

Case 3(Delete):

```
case "3" -> {
    System.out.println("Ingrese nombre del canal a eliminar");
    String targetChannel = br.readLine();
    AVLTree.Node target = tree.find(targetChannel);
    if(target != null){
        tree.delete(targetChannel);
        System.out.println("Canal eliminado");
    } else {
        System.out.println("Canal no encontrado");
    }
}
```

Ilustración 13: (Case 3)

En este caso, se le preguntará al usuario el nombre del canal que desea eliminar, este caso iniciará de la misma manera que el caso anterior, se creará una variable de nombre **“target”** de tipo **“Node”** que se hereda de la clase **“AVLTree”** que tendrá como valor lo ingresado por el usuario, después, esta variable se somete a un par de **“if”** que determinarán si el nombre del canal ingresado está dentro del árbol, si es así lo elimina y si no, arrojará por pantalla que no se encuentra el canal.

Conclusiones:

Durante el proceso de implementación y depuración del código, los procesos de depuración fueron mayoritariamente en base a la lógica y análisis del árbol, ya que una vez conocida la estructura más simple y básica junto a su funcionamiento pudo ser un poco más sencillo ell poder identificar a qué había que igualar las variables (o modificarlas) para que implementara un orden correcto al momento de imprimir, manteniendo en todo momento en consideración el objetivo de poder perfeccionar el correcto orden del árbol.

Por otro lado, la implementación de los métodos fue compleja debido a que la lógica y funcionamiento del árbol binario puede resultar ser difícil en ciertas situaciones, ya que se requiere hacer una investigación a fondo de sus modalidades, para así sacar el máximo provecho de sus funciones y poder encontrar el método más efectivo para lo que se desea llevar a cabo.

Bibliografía:

Para la implementación de las funciones se utilizó materia realizada en clases y la aclaración de la profesora Daniela Moreno.