GUÍA DE LABORATORIO 05

Tema: Arbol AVL

Nota

Estudiante	Escuela	Asignatura
Joel Isaias Condori Leon Carrasco Choque Arles Melvin Chara Condori Jean Carlo Hancco Soncco Vladimir Jaward	Escuela Profesional de Ingeniería de Sistemas	Estructura de Datos y Algoritmos Semestre: III Código: 1702224

Laboratorio	Tema	Duración
05	Arbol AVL	02 horas

Semestre académico	Fecha de inicio	Fecha de entrega
2023 - A	19 Junio 2023	26 Junio 2023

1. TAREA:

- Elabore un informe implementando Arboles AVL con toda la lista de operaciones search(), getMin(), getMax(), parent(), son(), insert(), remove().
- INPUT: Una sola palabra en mayusculas.
- OUTPUT: Se debe contruir el arbol AVL considerando el valor decimal de su codigo ascii.
- Luego, pruebe todas sus operaciones implementadas.
- Estudie la libreria Graph Stream para obtener una salida grafica de su implementacion.
- Utilice todas las recomendaciones dadas por el docente.

2. URL DEL REPOSITORIO EN GITHUB:

- Repositorio GITHUB: `https://github.com/carrascoArles/lab05-EDA.git`

3. DESARROLLO DE LA ACTIVIDAD:

3.1. Nodo AVL

La clase NodeAVL<E>representa un nodo en un árbol AVL. Cada nodo tiene un valor de tipo E, punteros a los nodos hijo izquierdo y derecho, y un campo bf que indica el factor de equilibrio del nodo.

- La clase es genérica, lo que significa que se puede utilizar para crear nodos que contengan cualquier tipo de objeto. El tipo de objeto se especifica cuando se crea una instancia de la clase.
- El constructor NodeAVL(E data, NodeAVL<E>left, NodeAVL<E>right) crea un nuevo nodo con un valor dado y un puntero a los nodos hijo izquierdo y derecho. El factor de equilibrio se establece en cero en el momento de la creación del nodo.
- El método getBf() devuelve el factor de equilibrio del nodo, que es la diferencia entre la altura del subárbol derecho y el subárbol izquierdo del nodo.
- Los métodos setData(E data), setLeft(NodeAVL<E>left), y setRight(NodeAVL<E>right) permiten establecer el valor y los punteros a los nodos hijo izquierdo y derecho del nodo, respectivamente.
- El método toString() devuelve una representación en cadena del nodo, que consiste en el valor del nodo y su factor de equilibrio entre paréntesis. Este método es útil para imprimir el árbol AVL en un formato legible.

3.2. Arbol AVL

A continuación tenemos la implementación de la estructura de datos para el árbol AVL en Java. Un árbol AVL es un árbol binario de búsqueda auto-balanceado donde las alturas de los dos subárboles hijos de cualquier nodo difieren como máximo en uno. El código proporciona métodos para insertar, buscar y eliminar elementos del árbol AVL.

La clase arbolAVL está parametrizada con un tipo genérico E que debe ser comparable, es decir, que debe implementar la interfaz Comparable<E>para poder comparar los elementos. Así mismo, hace uso de la clase NodeAVL<E>explicada anteriormente.

Esta clase usa los métodos:

- isEmpty(): Este método simplemente verifica si el árbol está vacío, es decir, si la raíz es igual a null. Si la raíz es nula, entonces el árbol está vacío y se devuelve true, de lo contrario se devuelve false.

Listing 1: Método isEmpty()

```
1   public boolean isEmpty() {
2       return this.root == null;
3   }
```

- getRoot(): Este método devuelve el valor de la raíz del árbol. Si la raíz es nula, entonces se lanza una excepción ExceptionNoFound.

Listing 2: Método getRoot()

```
1   public E getRoot() {
2       return this.root.getData();
3   }
```

- isLeaf(NodeAVL<E>current): Este método verifica si un nodo dado es una hoja, es decir, si no tiene hijos izquierdo ni derecho. La función devuelve true si el nodo es una hoja y false en caso contrario.

Listing 3: Método isLeaf(NodeAVL<E>current)

```
1   private boolean isLeaf(NodeAVL<E> current) { // elemento es hoja
```

```
        return current.getLeft() == null && current.getRight() == null;
    }
```

- parent(E x): Este método devuelve el padre del nodo que contiene el valor x. Para buscar el padre del nodo, se comienza la búsqueda desde la raíz del árbol y se desciende por el subárbol izquierdo o derecho según corresponda hasta encontrar el nodo que contiene el valor x. Si se encuentra el nodo, se devuelve su padre, de lo contrario se lanza una excepción ExceptionNoFound.

Listing 4: Método parent(E x)

```
    public NodeAVL<E> parent(E x) throws ExceptionNoFound {
        NodeAVL<E> aux = parent(x, this.root);
        if (aux == null)
            throw new ExceptionNoFound("Elemento no se encuentra en el arbol");
        System.out.println(aux.getData());
        return aux;
    }

    private NodeAVL<E> parent(E x, NodeAVL<E> current) throws ExceptionNoFound {
        if (current == null) {
            throw new ExceptionNoFound("El árbol está vacío");
        }

        NodeAVL<E> parent = null;
        NodeAVL<E> node = current;

        while (node != null) {
            int comparison = x.compareTo(node.getData());

            if (comparison == 0) {
                if (parent == null) {
                    throw new ExceptionNoFound("El nodo insertado es la raíz del árbol");
                }
                return parent;
            } else if (comparison < 0) {
                parent = node;
                node = node.getLeft();
            } else {
                parent = node;
                node = node.getRight();
            }
        }

        throw new ExceptionNoFound("El elemento no se encuentra en el árbol");
    }
```

- son(E dato): Este método devuelve una lista con los hijos del nodo que contiene el valor dato. Si el nodo no existe en el árbol, se lanza una excepción ExceptionNoFound.

Listing 5: Método son(E dato)

```
    public ArrayList<NodeAVL<E>> son(E dato) throws ExceptionNoFound {
        NodeAVL<E> nodo = search(dato);
        ArrayList<NodeAVL<E>> hijos = new ArrayList<>();
        if (nodo.getLeft() != null) {
            hijos.add(nodo.getLeft());
        }
```

```
        if (nodo.getRight() != null) {
            hijos.add(nodo.getRight());
        }
        return hijos;
    }
```

- insert(E x): Este método inserta un nuevo nodo con valor x en el árbol. Para insertar el nodo, se comienza la búsqueda desde la raíz del árbol y se desciende por el subárbol izquierdo o derecho según corresponda hasta encontrar el lugar adecuado para insertar el nuevo nodo. Si el valor x ya existe en el árbol, se lanza una excepción ExceptionNoFound.

Listing 6: Método insert(E x)

```
    public void insert(E x) throws ExceptionNoFound {
        this.root = insert(x, this.root);
        this.height = false;
    }

    private NodeAVL<E> insert(E x, NodeAVL<E> current) throws ExceptionNoFound {
        NodeAVL<E> res = current;
        if (current == null) {
            res = new NodeAVL<E>(x);
            this.height = true;
        } else {
            int resC = current.getData().compareTo(x);
            if (resC == 0)
                throw new ExceptionNoFound("El elemento ya se encuentra en el arbol");
            if (resC < 0) {
                res.setRight(insert(x, current.getRight()));
                if (this.height) {
                    switch (res.getBf()) {
                        case -1:
                            res.setBf(0); // -1+1
                            this.height = false;
                            break;
                        case 0:
                            res.setBf(1); // 0+1
                            break;
                        case 1: // res.setBf(2); 1+1
                            res = balanceToLeft(res);
                            this.height = false;
                            break;
                    }
                }

            } else { // resC > 0
                res.setLeft(insert(x, current.getLeft()));
                if (this.height) {
                    switch (res.getBf()) {
                        case -1: // res.setBf(-2); //-1-1
                            res = balanceToRight(res);
                            this.height = false;
                            break;
                        case 0:
                            res.setBf(-1); // 0-1
                            break;
                        case 1:
                            res.setBf(0); // 1-1
```

```
                        this.height = false;
                        break;
                    }
                }
            }
        }
        return res;
    }
```

- search(E x): Este método busca el nodo que contiene el valor x. Para buscar el nodo, se comienza la búsqueda desde la raíz del árbol y se desciende por el subárbol izquierdo o derecho según corresponda hasta encontrar el nodo que contiene el valor x. Si se encuentra el nodo, se devuelve, de lo contrario se lanza una excepción ExceptionNoFound.

Listing 7: Método search(E x)

```
    public NodeAVL<E> search(E x) throws ExceptionNoFound {
        NodeAVL<E> aux = search(x, this.root);
        if (aux == null) {
            throw new ExceptionNoFound("Elemento no se encuentra en el arbol");
        }
        return aux;
    }

    private NodeAVL<E> search(E x, NodeAVL<E> current) throws ExceptionNoFound {
        if (current == null) { // deja la recursividad si ya no hay elementos
            return null;
        } else {
            int resC = current.getData().compareTo(x);
            if (resC == 0) // deja la recursividad hasta encontrar el elemento
                return current;
            if (resC < 0)
                return search(x, current.getRight());
            else
                return search(x, current.getLeft());
        }
    }
```

- remove(E x): Este método elimina el nodo que contiene el valor x del árbol. Para eliminar el nodo, se comienza la búsqueda desde la raíz del árbol y se desciende por el subárbol izquierdo o derecho según corresponda hasta encontrar el nodo que contiene el valor x. Si el nodo no existe en el árbol, se lanza una excepción ExceptionNoFound. Si el nodo tiene dos hijos, se reemplaza el valor del nodo con el valor del sucesor inorden y se elimina el sucesor inorden. Si el nodo tiene un solo hijo o es una hoja, se elimina el nodo y se reorganiza el árbol para mantener el equilibrio de la estructura.

Listing 8: Método remove(E x)

```
    public void remove(E x) throws ExceptionNoFound {
        this.root = remove(x, this.root);
    }

    private NodeAVL<E> remove(E x, NodeAVL<E> current) throws ExceptionNoFound {
        NodeAVL<E> res = current;
        if (current == null) {
            throw new ExceptionNoFound("Elemento no se encuentra en el arbol");
        } else {
            int resC = current.getData().compareTo(x);
            if (resC < 0) {
```

```
            res.setRight(remove(x, current.getRight()));

        }
        else if (resC > 0) {
            res.setLeft(remove(x, current.getLeft()));

        }
        else {
            if (current.getLeft()!= null && current.getRight() != null) { // tiene ambos hijos (2)
                NodeAVL<E> aux = getMax(current.getLeft());
                       E datoAux = aux.getData();
                current.setLeft(remove(datoAux, current.getLeft()));
                       current.setData(datoAux);
            }
            else {
                if (isLeaf(current)) // es una hoja
                    res = null;
                else{ // solo tiene un hijo
                    res = current.getLeft() != null ? current.getLeft() : current.getRight();
                }
            }
        }
    }
    return res;

}
```

- getMin(): Este método devuelve el nodo con el valor mínimo del árbol. Para encontrar el nodo con el valor mínimo, se comienza la búsqueda desde la raíz del árbol y se desciende por el subárbol izquierdo hasta encontrar el nodo con el valor mínimo.

Listing 9: Método getMin()

```
public NodeAVL<E> getMin() {
    return getMin(root);
}

public NodeAVL<E> getMin(NodeAVL<E> nodo) {
    if (nodo.getLeft() == null) {
        return nodo;
    }
    return getMin(nodo.getLeft());
}
```

- getMax(): Este método devuelve el nodo con el valor máximo del árbol. Para encontrar el nodo con el valor máximo, se comienza la búsqueda desde la raíz del árbol y se desciende por el subárbol derecho hasta encontrar el nodo con el valor máximo.

Listing 10: Método getMax()

```
public NodeAVL<E> getMax() {
    return getMax(root);
}

public NodeAVL<E> getMax(NodeAVL<E> nodo) {
    if (nodo.getRight() == null) {
        return nodo;
```

```
8         }
9         return getMax(nodo.getRight());
10    }
```

- inOrden(): Este método imprime los valores del árbol en orden, es decir, en el orden en que se visitan los nodos en un recorrido inorden. Para hacer esto, se utiliza una implementación recursiva del recorrido inorden.

Listing 11: Método inOrden()

```
1     public void inOrden() {
2         if (isEmpty()) {
3             System.out.println("Arbol esta vacío ....");
4         } else {
5             inOrden(this.root);
6             System.out.println();
7         }
8     }
9
10    private void inOrden(NodeAVL<E> current) {
11        if (current.getLeft() != null) {
12            inOrden(current.getLeft());
13        }
14        System.out.print(current + ", ");
15        if (current.getRight() != null) {
16            inOrden(current.getRight());
17        }
18    }
```

4. SOLUCIÓN DEL CUESTIONARIO:

Explique como es el algoritmo que implemento para obtener el factor de equilibrio de un nodo.

- El algoritmo implementado para obtener el factor de equilibrio de un nodo en un árbol AVL se encuentra en el método getBalanceFactor(AVLNode node).

  Si el nodo dado es nulo (null), significa que no hay subárbol y su factor de equilibrio es 0. Por lo tanto, se devuelve 0.

  Si el nodo no es nulo, se calcula la altura de su subárbol izquierdo llamando al método getHeight(node.left). De manera similar, se calcula la altura del subárbol derecho llamando a getHeight(node.right).

  El factor de equilibrio se calcula restando la altura del subárbol derecho de la altura del subárbol izquierdo: getHeight(node.left) - getHeight(node.right).

  El valor resultante es el factor de equilibrio del nodo y se devuelve como resultado.

5. CONCLUSIONES:

- Los árboles AVL son una estructura de datos muy eficiente para almacenar y buscar información, especialmente cuando se necesita mantener el árbol balanceado y se espera realizar muchas operaciones de inserción y eliminación.
- La complejidad de tiempo para buscar un elemento en un árbol AVL es O(log n), lo que significa que la búsqueda es muy rápida incluso para árboles grandes.
- La principal desventaja de los árboles AVL es que pueden requerir más tiempo y recursos para realizar operaciones de inserción y eliminación que otros tipos de árboles, como los árboles binarios de búsqueda no balanceados.

6. REFERENCIAS:

- https://www.w3schools.com/java/default.asp
- https://graphstream-project.org/
- https://docs.oracle.com/middleware/1213/coherence/java-reference/com/tangosol/util/Tree.html
- https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/geometry/partitioning/utilities/AVLTree.html