Informe de Laboratorio 05

Tema: Arbol AVL

Nota

Estudiantes	Escuela	Asignatura
Condorios Yllapuma Jorge Cusilayme García Jose Mamani Mamani Alexis Valdivia Luna Carlo	Escuela Profesional de Ingeniería de Sistemas	Estructura de Datos Semestre: III Código: 1702124

Laboratorio	Tema	Duración
05	Arbol AVL	04 horas

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

1. Tarea

- Elabore un informe implementando Árboles AVL con toda la lista de operaciones search(), getMin(), getMax(), parent(), son(), insert(), remove().
- INPUT: Una sóla palabra en mayúsculas.
- OUTPUT: Se debe contruir el árbol AVL considerando el valor decimal de su código ascii.
- Luego, pruebe todas sus operaciones implementadas.
- Estudie la librería Graph Stream para obtener una salida gráfica de su implementación. Utilice todas las recomendaciones dadas por el docente.

2. Equipos, materiales y temas utilizados

- Windows 10 Home Single Language 64 bits (10.0, compilación 19045)
- VIM 9.0.
- Git 2.40.1.
- Cuenta en GitHub con el correo institucional.
- Java

3. URL de Repositorio Github

- URL del Repositorio GitHub para clonar o recuperar.
- `https://github.com/JorgeCY21/EDA_LAB_D`
- URL para el laboratorio 04 en el Repositorio GitHub.
- `https://github.com/JorgeCY21/EDA_LAB_D/tree/main/Lab_05`

4. Resolucion del ejercicio propuesto

4.1. Ejercicio 1

- Utilizar el tipo generico de NodeAvl.

Listing 1: Node.java

```
1
2  public class NodeAvl <E> {
3     E data;
4     NodeAvl<E> left, right;
5     int height;
6
7     NodeAvl(E data) {
8        this.data = data;
9        this.height = 1;
10    }
11
12    public E getData() {
13       return this.data;
14    }
15
16    public NodeAvl<E> getLeft() {
17       return this.left;
18    }
19
20    public NodeAvl<E> getRight() {
21       return this.right;
22    }
23 }
```

- La clase NodeAvl tiene un atributo data que representa los datos almacenados en el nodo.
- También tiene atributos left y right que representan los nodos hijos izquierdo y derecho, respectivamente.
- El atributo height almacena la altura del nodo en el árbol AVL.
- El constructor de la clase NodeAvl toma un parámetro data y lo asigna al atributo correspondiente, estableciendo la altura inicial en 1.

  Métodos:

  - **getData():** Devuelve el dato almacenado en el nodo.
  - **getLeft():** Devuelve el nodo hijo izquierdo.

- **getRight():** Devuelve el nodo hijo derecho.

- En resumen, la clase NodeAvl representa un nodo en un árbol AVL y proporciona métodos para acceder a sus datos y nodos hijos. Este código puede ser utilizado en la implementación de un árbol AVL para realizar operaciones de inserción, eliminación y búsqueda de manera eficiente.

- Se utiliza el NodeAvl en la implementación de la clase TreeAVL.

Listing 2: LinkedList.java

```
1
2  class TreeAVL<E extends Comparable<E>> {
3     private NodeAvl<E> root;
4
5     public void insert(E data) {
6        this.root = insertNode(this.root, data);
7     }
8
9     private NodeAvl<E> insertNode(NodeAvl<E> node, E data) {
10       if (node == null) {
11          return new NodeAvl<E>(data);
12       }
13
14       if (data.compareTo(node.data) < 0) {
15          node.left = insertNode(node.left, data);
16       } else if (data.compareTo(node.data) > 0) {
17          node.right = insertNode(node.right, data);
18       } else {
19          return node;
20       }
21
22       node.height = 1 + Math.max(height(node.left), height(node.right));
23
24       int balance = getBalance(node);
25
26       if (balance > 1 && data.compareTo(node.left.data) < 0) {
27          return rotateRight(node);
28       }
29
30       if (balance < -1 && data.compareTo(node.right.data) > 0) {
31          return rotateLeft(node);
32       }
33
34       if (balance > 1 && data.compareTo(node.left.data) > 0) {
35          node.left = rotateLeft(node.left);
36          return rotateRight(node);
37       }
38
39       if (balance < -1 && data.compareTo(node.right.data) < 0) {
40          node.right = rotateRight(node.right);
41          return rotateLeft(node);
42       }
43
44       return node;
45    }
46
```

```
47    public void remove(E data) {
48       this.root = removeNode(this.root, data);
49    }
50
51    private NodeAvl<E> removeNode(NodeAvl<E> node, E data) {
52       if (node == null) {
53          return node;
54       }
55
56       if (data.compareTo(node.data) < 0) {
57          node.left = removeNode(node.left, data);
58       } else if (data.compareTo(node.data) > 0) {
59          node.right = removeNode(node.right, data);
60       } else {
61          if (node.left == null || node.right == null) {
62             NodeAvl<E> temp = null;
63             if (temp == node.left) {
64                temp = node.right;
65             } else {
66                temp = node.left;
67             }
68
69             if (temp == null) {
70                temp = node;
71                node = null;
72             } else {
73                node = temp;
74             }
75          } else {
76             NodeAvl<E> temp = minValueNode(node.right);
77             node.data = temp.data;
78             node.right = removeNode(node.right, temp.data);
79          }
80       }
81
82       if (node == null) {
83          return node;
84       }
85
86       node.height = 1 + Math.max(height(node.left), height(node.right));
87
88       int balance = getBalance(node);
89
90       if (balance > 1 && getBalance(node.left) >= 0) {
91          return rotateRight(node);
92       }
93
94       if (balance < -1 && getBalance(node.right) <= 0) {
95          return rotateLeft(node);
96       }
97
98       if (balance > 1 && getBalance(node.left) < 0) {
99          node.left = rotateLeft(node.left);
100         return rotateRight(node);
101      }
102
```

```
103        if (balance < -1 && getBalance(node.right) > 0) {
104           node.right = rotateRight(node.right);
105           return rotateLeft(node);
106        }
107
108        return node;
109     }
110
111     public E getMin() {
112        if (root == null) {
113           return null;
114        }
115        NodeAvl<E> minNode = getMinNode(root);
116        return minNode.data;
117     }
118
119     private NodeAvl<E> minValueNode(NodeAvl<E> node) {
120        NodeAvl<E> current = node;
121         while (current.left != null) {
122             current = current.left;
123         }
124         return current;
125     }
126
127     private NodeAvl<E> getMinNode(NodeAvl<E> node) {
128        NodeAvl<E> current = node;
129        while (current.left != null) {
130           current = current.left;
131        }
132        return current;
133     }
134
135     public E getMax() {
136        if (root == null) {
137           return null;
138        }
139        NodeAvl<E> maxNode = getMaxNode(root);
140        return maxNode.data;
141     }
142
143     private NodeAvl<E> getMaxNode(NodeAvl<E> node) {
144        NodeAvl<E> current = node;
145        while (current.right != null) {
146           current = current.right;
147        }
148        return current;
149     }
150
151     public boolean search(E data) {
152        return searchNode(root, data);
153     }
154
155     private boolean searchNode(NodeAvl<E> node, E data) {
156        if (node == null) {
157           return false;
158        }
```

```
159
160         if (data.compareTo(node.data) < 0) {
161             return searchNode(node.left, data);
162         } else if (data.compareTo(node.data) > 0) {
163             return searchNode(node.right, data);
164         } else {
165             return true;
166         }
167     }
168
169     private int height(NodeAvl<E> node) {
170         if (node == null) {
171             return 0;
172         }
173         return node.height;
174     }
175
176     private int getBalance(NodeAvl<E> node) {
177         if (node == null) {
178             return 0;
179         }
180         return height(node.left) - height(node.right);
181     }
182
183     private NodeAvl<E> rotateRight(NodeAvl<E> y) {
184         NodeAvl<E> x = y.left;
185         NodeAvl<E> T2 = x.right;
186
187         x.right = y;
188         y.left = T2;
189
190         y.height = Math.max(height(y.left), height(y.right)) + 1;
191         x.height = Math.max(height(x.left), height(x.right)) + 1;
192
193         return x;
194     }
195
196     private NodeAvl<E> rotateLeft(NodeAvl<E> x) {
197         NodeAvl<E> y = x.right;
198         NodeAvl<E> T2 = y.left;
199
200         y.left = x;
201         x.right = T2;
202
203         x.height = Math.max(height(x.left), height(x.right)) + 1;
204         y.height = Math.max(height(y.left), height(y.right)) + 1;
205
206         return y;
207     }
208
209     public NodeAvl<E> getRoot() {
210         return this.root;
211     }
212 }
```

- La clase TreeAVL representa un árbol AVL y proporciona métodos para insertar, eliminar y

buscar elementos en el árbol. Aquí está una descripción del código en tercera persona:

- La clase TreeAVL tiene un atributo privado root que representa el nodo raíz del árbol AVL. **Métodos:**
  - **insert(E data):** Inserta un elemento en el árbol AVL. Este método llama al método privado insertNode para realizar la inserción.
  - **insertNode(NodeAvl ¡E¿node, E data):** Método privado que realiza la inserción de un nodo en el árbol AVL. Recibe un nodo y un dato a insertar. Si el nodo es nulo, crea un nuevo nodo con el dato. Luego, compara el dato con el dato del nodo actual y decide si debe insertarse en el subárbol izquierdo o derecho. Después de la inserción, se actualiza la altura del nodo y se realiza el equilibrado del árbol si es necesario, utilizando las rotaciones adecuadas.
  - **remove(E data):** Elimina un elemento del árbol AVL. Este método llama al método privado removeNode para realizar la eliminación.
  - **removeNode(NodeAvl¡E¿node, E data):** Método privado que realiza la eliminación de un nodo del árbol AVL. Recibe un nodo y el dato a eliminar. Si el nodo es nulo, retorna el mismo nodo. Luego, compara el dato con el dato del nodo actual y decide si debe eliminarlo del subárbol izquierdo o derecho. Dependiendo de la situación, se manejan diferentes casos para garantizar la conservación de las propiedades del árbol AVL. Se actualiza la altura del nodo y se realiza el equilibrado si es necesario.
  - **getMin():** Devuelve el elemento mínimo (el menor) en el árbol AVL. Utiliza el método privado getMinNode para encontrar el nodo mínimo y devuelve el dato contenido en él.
  - **getMinNode(NodeAvl¡E¿node):** Método privado que encuentra el nodo mínimo en el árbol AVL. Recibe un nodo y realiza un recorrido hacia la izquierda hasta encontrar el nodo más a la izquierda del subárbol.
  - **getMax():** Devuelve el elemento máximo (el mayor) en el árbol AVL. Utiliza el método privado getMaxNode para encontrar el nodo máximo y devuelve el dato contenido en él.
  - **getMaxNode(NodeAvl¡E¿node):** Método privado que encuentra el nodo máximo en el árbol AVL. Recibe un nodo y realiza un recorrido hacia la derecha hasta encontrar el nodo más a la derecha del subárbol.
  - **search(E data):** Busca un elemento en el árbol AVL. Utiliza el método privado searchNode para realizar la búsqueda.
  - **searchNode(NodeAvl¡E¿node, E data):** Método privado que realiza la búsqueda de un dato en el árbol AVL. Recibe un nodo y el dato a buscar. Realiza una comparación entre el dato y el dato del nodo actual y decide si debe buscar en el subárbol izquierdo o derecho. Si encuentra el dato, retorna verdadero; de lo contrario, retorna falso.
  - **height(NodeAvl¡E¿node):** Método privado que devuelve la altura de un

Listing 3: Test.java

```
1  import java.util.Scanner;
2  import org.graphstream.graph.Graph;
3  import org.graphstream.graph.Node;
4  import org.graphstream.graph.implementations.SingleGraph;
5
6  public class Test {
7
8     public static void main(String[] args) {
9
```

```
10      Scanner scan = new Scanner(System.in);
11
12      TreeAVL<Integer> treeAvl = new TreeAVL<>();
13      String input = scan.next();
14
15      for(char c: input.toCharArray()) {
16         int asciiValue = (int)c;
17         treeAvl.insert(asciiValue);
18      }
19
20      System.out.println("dibujando...");
21
22      Graph graph = new SingleGraph("Tree AVL");
23        graph.setAttribute("ui.stylesheet", "node { size: 20px; text-size: 15px; }");
24        graph.setStrict(false);
25        System.setProperty("org.graphstream.ui",
              "org.graphstream.ui.swing.SwingGraphRenderer");
26
27        visualizeTree(treeAvl.getRoot(), graph);
28
29        graph.display();
30
31      System.out.println("Minimum value: " + treeAvl.getMin()); // Valor mínimo: 10
32      System.out.println("Maximum value: " + treeAvl.getMax()); // Valor máximo: 50
33
34      System.out.println("Buscar 30: " + treeAvl.search(30)); // Búsqueda de 30: true
35      System.out.println("Buscar 60: " + treeAvl.search(60)); // Búsqueda de 60: false
36
37      treeAvl.remove(30);
38
39      System.out.println("Buscar 30 después de eliminarlo: " + treeAvl.search(30)); //
            Búsqueda de 30 después de
40                                                          // eliminarlo: false
41
42
43
44  }
45
46  private static void visualizeTree(NodeAvl<Integer> node, Graph graph) {
47      if (node != null) {
48            graph.addNode(String.valueOf(node.getData())).setAttribute("ui.label",
                  node.getData());
49            if (node.getLeft() != null) {
50                visualizeTree(node.getLeft(), graph);
51                graph.addEdge(node.getData() + "-" + node.getLeft().getData(),
                      String.valueOf(node.getData()), String.valueOf(node.getLeft().getData()));
52            }
53            if (node.getRight() != null) {
54                visualizeTree(node.getRight(), graph);
55                graph.addEdge(node.getData() + "-" + node.getRight().getData(),
                      String.valueOf(node.getData()), String.valueOf(node.getRight().getData()));
56            }
57        }
58
59  }
60
```

```
61 }
```

- La clase Test contiene un método main que realiza las siguientes acciones:
  - Crea un objeto Scanner para leer la entrada del usuario.
  - Crea un objeto TreeAVL¡Integer¿para almacenar los elementos del árbol AVL.
  - Lee una cadena de texto desde la entrada y la convierte en valores ASCII.
  - Inserta los valores en el árbol AVL.
  - Imprime un mensaje indicando que se va a dibujar el árbol.
  - Crea un objeto Graph utilizando GraphStream y configura propiedades visuales.
  - Visualiza el árbol AVL en el grafo.
  - Muestra el grafo en una ventana emergente.
  - Obtiene y muestra el valor mínimo del árbol AVL.
  - Obtiene y muestra el valor máximo del árbol AVL.
  - Realiza búsquedas en el árbol AVL y muestra los resultados.
  - Elimina un valor del árbol AVL.
  - Realiza otra búsqueda y muestra el resultado.
- El método visualizeTree es un método auxiliar que visualiza recursivamente el árbol AVL en el grafo.

5. Preguntas

5.1. ¿Explique como es el algoritmo que implementó para obtener el factor de equilibrio de un nodo?

- El algoritmo implementado para obtener el factor de equilibrio de un nodo en el árbol AVL se realiza de la siguiente manera:
  - En el método insertNode, después de insertar un nuevo nodo en el árbol AVL, se actualiza la altura del nodo actual. La altura se calcula como el máximo entre la altura de su subárbol izquierdo y su subárbol derecho, más 1.
  - A continuación, se obtiene el factor de equilibrio del nodo llamando al método getBalance. El factor de equilibrio se calcula restando la altura del subárbol derecho del nodo a la altura del subárbol izquierdo.
  - Si el factor de equilibrio es mayor que 1 y el valor a insertar es menor que el valor del nodo izquierdo, se realiza una rotación hacia la derecha en el nodo actual.
  - Si el factor de equilibrio es menor que -1 y el valor a insertar es mayor que el valor del nodo derecho, se realiza una rotación hacia la izquierda en el nodo actual.
  - Si el factor de equilibrio es mayor que 1 y el valor a insertar es mayor que el valor del nodo izquierdo, se realiza una rotación hacia la izquierda en el nodo izquierdo, seguida de una rotación hacia la derecha en el nodo actual.
  - Si el factor de equilibrio es menor que -1 y el valor a insertar es menor que el valor del nodo derecho, se realiza una rotación hacia la derecha en el nodo derecho, seguida de una rotación hacia la izquierda en el nodo actual.

- El algoritmo para obtener el factor de equilibrio se aplica de manera similar en el método removeNode cuando se elimina un nodo del árbol AVL. Después de realizar la eliminación, se actualiza la altura del nodo actual y se calcula el factor de equilibrio. Se aplican las rotaciones correspondientes según el caso para mantener el equilibrio del árbol AVL.
- Además, se utilizan los métodos rotateRight y rotateLeft para realizar las rotaciones hacia la derecha y hacia la izquierda, respectivamente, en los nodos del árbol AVL.
- La clase TreeAVL también proporciona otros métodos como getMin, getMax y search para obtener el valor mínimo, valor máximo y realizar búsquedas en el árbol AVL, respectivamente. Estos métodos utilizan operaciones de comparación para recorrer el árbol y encontrar los valores deseados.
- En resumen, el algoritmo para obtener el factor de equilibrio en un árbol AVL se basa en el cálculo de las alturas de los subárboles y la resta de estas alturas para determinar el equilibrio del nodo. Luego se aplican las rotaciones necesarias para mantener el equilibrio del árbol después de la inserción o eliminación de nodos.

6. Conclusiones

- La implementación de una cola de prioridad utilizando un heap proporciona un tiempo de ejecución eficiente para las operaciones de inserción y eliminación de elementos con mayor prioridad.
- El uso de estructuras de datos genéricas y programación orientada a objetos permite crear implementaciones flexibles y reutilizables.
- El diseño de clases y métodos bien encapsulados promueve un código limpio y mantenible.
- La comprensión de las propiedades y operaciones del heap es fundamental para implementar correctamente una cola de prioridad.
- En este ejercicio, se ha implementado una cola de prioridad utilizando un heap como estructura de datos subyacente. La implementación del heap permite realizar las operaciones de inserción, eliminación y acceso a los elementos con mayor y menor prioridad de manera eficiente. El uso de una clase genérica y la implementación de la interfaz Comparable proporcionan flexibilidad para trabajar con diferentes tipos de elementos y prioridades.
- La implementación de una cola de prioridad basada en un heap es una estrategia poderosa y eficiente para resolver problemas que implican la gestión de elementos con diferentes niveles de prioridad. Al comprender los conceptos y técnicas presentadas en este ejercicio, se puede aplicar este conocimiento a una amplia gama de problemas en los que se requiere un manejo eficiente de la prioridad.
- Informe a detalle en:
- `https://docs.google.com/document/d/1jAOoSbbO_PYWXzPXPUOXgSOeuRT48e6vSWjYnETln9Y/edit?usp=sharing`

7. Referencias

- `https://www.geeksforgeeks.org/introduction-to-avl-tree/`
- `https://www.geeksforgeeks.org/insertion-in-an-avl-tree/`
- `https://www.geeksforgeeks.org/deletion-in-an-avl-tree/`

- https://www.javatpoint.com/avl-tree
- https://docs.oracle.com/javase/tutorial/java/generics/types.html
- https://algorithmtutor.com/Data-Structures/Tree/AVL-Trees/