**AVL TREE**

1. package avl;

2.

3. import java.util.ArrayList;

4. import java.util.Scanner; // Import the Scanner class to read user input

5.

6. class Node { // Node class

7.     int key, height; // Node attributes

8.     Node left, right; // Node children

9.

10.     Node(int value) { // Node constructor

11.         key = value; // The key is the value of the node

12.         height = 1; // The height of a new node is 1

13.         left = right = null; // The left and right children of a new node are null

14.     } // End of Node constructor

15. } // End of Node class

16.

17. public class AVLTree { // AVLTree class

18.     Node root; // Root of the AVL Tree

19.

20.     /\*\*

21.      \* Method to get the height of a node in the AVL Tree

22.      \* @param node The node to get the height

23.      \* @return The height of the node

24.      \*/

25.     int height(Node node) { // Method to get the height of a node

26.         if (node == null) // If the node is null, return 0

27.             return 0; // If the node is null, return 0

28.         return node.height; // Return the height of the node

29.     } // End of height method

30.

31.     /\*\*

32.      \* Method to get the balance factor of a node in the AVL Tree (left height - right height)

33.      \* @param node The node to get the balance factor

34.      \* @return The balance factor of the node

35.      \*/

36.     int balanceFactor(Node node) { // Method to get the balance factor of a node

37.         if (node == null) // If the node is null, return 0

38.             return 0; // If the node is null, return 0

39.         return height(node.left) - height(node.right); // Return the balance factor of the node

40.     } // End of balanceFactor method

41.

42.     /\*\*

43.      \* Method to perform a right rotation in the AVL Tree

44.      \* @param y The node to perform the right rotation

45.      \* @return The new root of the AVL Tree

46.      \*/

47.     Node rightRotate(Node y) { // Method to perform a right rotation

48.         Node x = y.left; // Get the left child of the node

49.         Node T2 = x.right; // Perform the rotation

50.

51.         x.right = y; // Perform the rotation

52.         y.left = T2; // Perform the rotation

53.

54.         y.height = Math.max(height(y.left), height(y.right)) + 1; // Update the height of the nodes

55.         x.height = Math.max(height(x.left), height(x.right)) + 1; // Update the height of the nodes

56.

57.         return x; // Return the new root

58.     } // End of rightRotate method

59.

60.     /\*\*

61.      \* Method to perform a left rotation in the AVL Tree

62.      \* @param x The node to perform the left rotation

63.      \* @return The new root of the AVL Tree

64.      \*/

65.     Node leftRotate(Node x) { // Method to perform a left rotation

66.         Node y = x.right; // Get the right child of the node

67.         Node T2 = y.left; // Perform the rotation

68.

69.         y.left = x; // Perform the rotation

70.         x.right = T2; // Perform the rotation

71.

72.         x.height = Math.max(height(x.left), height(x.right)) + 1; // Update the height of the nodes

73.         y.height = Math.max(height(y.left), height(y.right)) + 1; // Update the height of the nodes

74.

75.         return y; // Return the new root

76.     } // End of leftRotate method

77.

78.     /\*\*

79.      \* Method to get the AVL Tree as array

80.      \* @return The AVL Tree as array

81.      \*/

82.     int[] toArray() { // Method to get the AVL Tree as array

83.         ArrayList<Integer> list = new ArrayList<>(); // Create an ArrayList to store the values

84.         toArrayHelper(root, list); // Call the helper method

85.

86.         int[] array = new int[list.size()]; // Create an array to store the values

87.         for (int i = 0; i < list.size(); i++) { // For loop to store the values

88.             array[i] = list.get(i).intValue(); // Store the value in the array

89.         } // End of for loop

90.

91.         return array; // Return the array

92.     } // End of toArray method

93.

94.     /\*\*

95.      \* Helper method to get the AVL Tree as array

96.      \* @param node The node to get the AVL Tree as array

97.      \* @param list The ArrayList to store the values

98.      \*/

99.     void toArrayHelper(Node node, ArrayList<Integer> list) { // Helper method to get the AVL Tree as array

100.         if (node == null) { // If the node is null, return

101.             return; // If the node is null, return

102.         } // End of if statement

103.

104.         toArrayHelper(node.left, list); // Call the helper method

105.         list.add(node.key); // Add the value to the ArrayList

106.         toArrayHelper(node.right, list); // Call the helper method

107.     } // End of toArrayHelper method

108.

109.     /\*\*

110.      \* Method to insert a new node in the AVL Tree and perform the rotations when necessary

111.      \* @param node The node to insert the new node

112.      \* @param key The value of the new node

113.      \* @return The new root of the AVL Tree

114.      \*/

115.     Node insert(Node node, int key) { // Method to insert a new node in the AVL Tree

116.         if (node == null) // If the node is null, return a new node

117.             return new Node(key); // If the node is null, return a new node

118.

119.         if (key < node.key) // If the key is less than the node key, insert the new node in the left subtree

120.             node.left = insert(node.left, key); // If the key is less than the node key, insert the new node in the left subtree

121.         else if (key > node.key) // If the key is greater than the node key, insert the new node in the right subtree

122.             node.right = insert(node.right, key); // If the key is greater than the node key, insert the new node in the right subtree

123.         else // If the key is equal to the node key, return the node

124.             return node; // If the key is equal to the node key, return the node

125.

126.         node.height = 1 + Math.max(height(node.left), height(node.right)); // Update the height of the node

127.

128.         int balance = balanceFactor(node); // Get the balance factor of the node

129.

130.         if (balance > 1) { // If the balance factor is greater than 1, the node is unbalanced

131.             if (key < node.left.key) { // If the balance factor is greater than 1, the node is unbalanced

132.                 System.out.println("Right Rotation at Node: " + node.key); // Print the message

133.                 return rightRotate(node); // Rotate the node to the right

134.             } else { // If the balance factor is greater than 1, the node is unbalanced

135.                 System.out.println("Left-Right Rotation at Node: " + node.key); // Print the message

136.                 node.left = leftRotate(node.left); // Rotate the node to the left

137.                 return rightRotate(node);

138.             } // End of if statement

139.         } // End of if statement

140.

141.         if (balance < -1) { // If the balance factor is less than -1, the node is unbalanced

142.             if (key > node.right.key) { // If the balance factor is less than -1, the node is unbalanced

143.                 System.out.println("Left Rotation at Node: " + node.key); // Print the message

144.                 return leftRotate(node); // Rotate the node to the left

145.             } else { // If the balance factor is less than -1, the node is unbalanced

146.                 System.out.println("Right-Left Rotation at Node: " + node.key); // Print the message

147.                 node.right = rightRotate(node.right); // Rotate the node to the right

148.                 return leftRotate(node); // Rotate the node to the left

149.             } // End of if statement

150.         } // End of if statement

151.

152.         return node; // Return the node

153.     } // End of insert method

154.

155.     /\*\*

156.      \* Method to delete a node in the AVL Tree and perform the rotations when necessary

157.      \* @param root The root of the AVL Tree

158.      \* @param key The value of the node to be deleted

159.      \* @return The new root of the AVL Tree

160.      \*/

161.     Node delete(Node root, int key) { // Method to delete a node in the AVL Tree

162.         if (root == null) // If the root is null, return the root

163.             return root; // If the root is null, return the root

164.

165.         if (key < root.key) // If the key is less than the root key, delete the node in the left subtree

166.             root.left = delete(root.left, key); // If the key is less than the root key, delete the node in the left subtree

167.

168.         else if (key > root.key) // If the key is greater than the root key, delete the node in the right subtree

169.             root.right = delete(root.right, key); // If the key is greater than the root key, delete the node in the right subtree

170.

171.         else { // If the key is equal to the root key, delete the node

172.             Node temp = null; // Create a temporary node

173.             if((root.left == null) || (root.right == null)) { // If the node has one child or no child

174.                 //Node temp = null; // Create a temporary node

175.

176.                 if (temp == root.left) // If the node has one child or no child

177.                     temp = root.right; // The temporary node is the right child

178.                 else // If the node has one child or no child

179.                     temp = root.left; // The temporary node is the left child

180.

181.                 if (temp == null) { // If the node has no child

182.                     temp = root; // The temporary node is the root

183.                     root = null; // The root is null

184.                 } else // If the node has no child

185.                     root = temp; // The root is the temporary node

186.             } else { // If the node has two children

187.                 temp = minValueNode(root.right); // Get the node with the minimum value in the right subtree

188.                 root.key = temp.key; // Replace the root key with the minimum value

189.                 root.right = delete(root.right, temp.key); // Delete the node with the minimum value

190.             } // End of if statement

191.         } // End of if statement

192.

193.         if (root == null) // If the root is null, return the root

194.             return root; // Return the root

195.

196.         root.height = 1 + Math.max(height(root.left), height(root.right)); // Update the height of the node

197.

198.         int balance = balanceFactor(root); // Get the balance factor of the node

199.

200.         if (balance > 1 && balanceFactor(root.left) >= 0) { // If the balance factor is greater than 1, the node is unbalanced

201.             System.out.println("Right Rotation at Node: " + root.key); // Print the message

202.             return rightRotate(root); // Rotate the node to the right

203.         } // End of if statement

204.

205.         if (balance > 1 && balanceFactor(root.left) < 0) { // If the balance factor is greater than 1, the node is unbalanced

206.             System.out.println("Left-Right Rotation at Node: " + root.key); // Print the message

207.             root.left = leftRotate(root.left); // Rotate the node to the left

208.             return rightRotate(root); // Rotate the node to the right

209.         } // End of if statement

210.

211.         if (balance < -1 && balanceFactor(root.right) <= 0) { // If the balance factor is less than -1, the node is unbalanced

212.             System.out.println("Left Rotation at Node: " + root.key); // Print the message

213.             return leftRotate(root); // Rotate the node to the left

214.         }

215.

216.         if (balance < -1 && balanceFactor(root.right) > 0) { // If the balance factor is less than -1, the node is unbalanced

217.             System.out.println("Right-Left Rotation at Node: " + root.key); // Print the message

218.             root.right = rightRotate(root.right); // Rotate the node to the right

219.             return leftRotate(root); // Rotate the node to the left

220.         } // End of if statement

221.

222.         return root; // Return the root

223.     } // End of delete method

224.

225.     /\*\*

226.      \* Method to get the node with the minimum value in the AVL Tree (leftmost node)

227.      \* @param node The root of the AVL Tree

228.      \* @return The node with the minimum value

229.      \*/

230.     Node minValueNode(Node node) { // Method to get the node with the minimum value

231.         Node current = node; // Get the root of the AVL Tree

232.

233.         while (current.left != null) // While the left child is not null

234.             current = current.left; // Get the left child

235.

236.         return current; // Return the node with the minimum value

237.     } // End of minValueNode method

238.

239.     void printTree(Node root, String indent, boolean last) { // Method to print the AVL Tree

240.         if (root != null) { // If the root is not null

241.             System.out.print(indent); // Print the indent

242.

243.             if (last) { // If the node is the last node

244.                 System.out.print("R----"); // Print the message

245.                 indent += "     "; // Update the indent

246.             } else { // If the node is not the last node

247.                 System.out.print("L----"); // Print the message

248.                 indent += "|    ";  // Update the indent

249.             } // End of if statement

250.

251.             System.out.println("Key: " + root.key + " Height: " + root.height + " Balance Factor: " + balanceFactor(root) + "\n"); // Print the node information

252.

253.             printTree(root.left, indent, false); // Print the left subtree

254.             printTree(root.right, indent, true); // Print the right subtree

255.         } // End of if statement

256.     } // End of printTree method

257.

258.     /\*\*

259.      \* Main method

260.      \* @param args The command line arguments

261.      \*/

262.     public static void main(String[] args) { // Main method

263.         Scanner scanner = new Scanner(System.in); // Create a Scanner object

264.         AVLTree tree = new AVLTree(); // Create an AVL Tree

265.         System.out.print("Type the number of elements of the tree, followed by the values "

266.                             + "to be inserted (Example: 5 10 25 18 41 3): "); // Print the prompt to the user

267.         String[] input = scanner.nextLine().split(" "); // Get the user input

268.

269.         int[] values = new int[Integer.parseInt(input[0])]; // Create an array to store the values

270.

271.         for (int i = 1; i < input.length; i++) { // For loop to store the values

272.             values[i - 1] = Integer.parseInt(input[i]); // Store the value in the array

273.         } // End of for loop

274.

275.         for (int value : values) { // For each value

276.             tree.root = tree.insert(tree.root, value); // Insert the value in the AVL Tree

277.             System.out.println("Balanced AVL Tree after inserting " + value + ":"); // Print the message

278.             tree.printTree(tree.root, "", true); // Print the AVL Tree

279.             System.out.println("------------------------------"); // Print the message

280.         } // End of for loop

281.

282.         while (true) { // While loop to add or remove nodes

283.             System.err.print("Enter (A) to add or (R) to remove a node, (P) to print the tree or (E) to exit: "); // Print the prompt to the user

284.             char choice = scanner.next().charAt(0); // Get the user input

285.

286.             if (choice == 'E' || choice == 'e') { // If the user wants to exit

287.                 break; // Break the loop

288.             } // End of if statement

289.

290.             if (choice == 'A' || choice == 'a') { // If the user wants to add a node

291.                 System.out.print("Enter the value to be inserted: "); // Print the prompt to the user

292.                 int value = scanner.nextInt(); // Get the user input

293.                 tree.root = tree.insert(tree.root, value); // Insert the value in the AVL Tree

294.                 System.out.println("Balanced AVL Tree after inserting " + value + ":"); // Print the message

295.                 tree.printTree(tree.root, "", true); // Print the AVL Tree

296.             } else if (choice == 'R' || choice == 'r') { // If the user wants to remove a node

297.                 System.out.print("Enter the value to be removed: "); // Print the prompt to the user

298.                 int value = scanner.nextInt(); // Get the user input

299.                 tree.root = tree.delete(tree.root, value); // Delete the value in the AVL Tree

300.                 System.out.println("Balanced AVL Tree after removing " + value + ":"); // Print the message

301.                 tree.printTree(tree.root, "", true); // Print the AVL Tree

302.             } else if(choice == 'P' || choice == 'p') { // If the user wants to print the AVL Tree

303.                 System.out.print("AVL Tree as Array: "); // Print the message

304.                 int[] array = tree.toArray(); // Get the AVL Tree as array

305.                 for (int value : array) { // For each value

306.                     System.out.print(value + " "); // Print the value

307.                 } // End of for loop

308.                 System.out.println(); // Print a new line

309.             } else { // If the user enters an invalid option

310.                 System.out.println("Invalid option!"); // Print the message

311.             } // End of if statement

312.

313.             //tree.printTree(tree.root, "", true); // Print the AVL Tree

314.         } // End of while loop

315.

316.         scanner.close(); // Close the scanner

317.     } // End of main method

318. } // End of AVLTree class

319.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**EXPLICAÇÃO DO ALGORITMO**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Classe Node:**
   * Representa um nó da árvore AVL. Cada nó possui um valor (**key**), altura (**height**), e referências para os nós à esquerda (**left**) e à direita (**right**).
2. **Classe AVLTree:**
   * Possui um ponteiro para a raiz da árvore.
3. **Métodos height e balanceFactor:**
   * **height(Node node)**: Retorna a altura de um nó. Se o nó for nulo, a altura é considerada 0.
   * **balanceFactor(Node node)**: Calcula o fator de equilíbrio de um nó, que é a diferença entre as alturas das subárvores esquerda e direita.
4. **Métodos de rotação (rightRotate e leftRotate):**
   * Realizam operações de rotação simples (direita ou esquerda) para manter a propriedade de árvore AVL.
5. **Método toArray:**
   * Converte a árvore AVL em um array. Usa um método auxiliar (**toArrayHelper**) para percorrer a árvore em ordem e adicionar os valores ao ArrayList.
6. **Método insert:**
   * Insere um novo valor na árvore AVL. Após a inserção, verifica o fator de equilíbrio e realiza rotações, se necessário, para manter a árvore balanceada.
7. **Método delete:**
   * Remove um valor da árvore AVL. Após a remoção, verifica o fator de equilíbrio e realiza rotações, se necessário, para manter a árvore balanceada.
8. **Método minValueNode:**
   * Encontra o nó com o menor valor na árvore.
9. **Método printTree:**
   * Imprime a árvore de forma visual, mostrando a chave, altura e fator de equilíbrio de cada nó.
10. **Método main:**

* Inicializa uma árvore AVL.
* Recebe a entrada do usuário para construir a árvore.
* Permite ao usuário adicionar, remover, imprimir a árvore ou sair do programa interativamente.

O código é interativo e permite que o usuário construa uma árvore AVL, insira e remova elementos, visualize a árvore de forma gráfica e converta a árvore em um array. Durante as operações de inserção e remoção, o código realiza rotações conforme necessário para manter a propriedade de árvore AVL.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**MÉTODOS PARA MANIPULAÇÃO DE ALTURA E FATOR DE EQUILÍBRIO**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Método height(Node node):**
   * Este método retorna a altura de um nó na árvore. Se o nó for nulo, a altura é considerada 0. A altura de um nó é a altura da subárvore mais alta entre suas subárvores esquerda e direita, mais 1.
2. **Método balanceFactor(Node node):**
   * Calcula o fator de equilíbrio de um nó, que é a diferença entre as alturas da subárvore esquerda e da subárvore direita. O fator de equilíbrio é utilizado para determinar se uma rotação é necessária para manter a propriedade da árvore AVL. Se o fator de equilíbrio for positivo, a subárvore esquerda é mais alta; se for negativo, a subárvore direita é mais alta.
3. **Método rightRotate(Node y):**
   * Realiza uma rotação para a direita em torno do nó **y**. Esta operação é utilizada para corrigir um desequilíbrio quando a subárvore esquerda de um nó está mais alta que a subárvore direita. A rotação envolve modificar os ponteiros dos nós envolvidos e atualizar as alturas dos nós afetados.
4. **Método leftRotate(Node x):**
   * Realiza uma rotação para a esquerda em torno do nó **x**. Semelhante à rotação direita, essa operação corrige um desequilíbrio quando a subárvore direita de um nó está mais alta que a subárvore esquerda.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**MÉTODOS PARA INSERÇÃO E REMOÇÃO**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. **Método insert(Node node, int key):**
   * Insere um novo nó na árvore com a chave **key**. Se a árvore estiver vazia, cria um novo nó. Caso contrário, insere recursivamente na subárvore esquerda ou direita, dependendo do valor da chave. Após a inserção, atualiza a altura do nó e verifica o fator de equilíbrio. Se necessário, realiza rotações para manter a árvore balanceada.
2. **Método delete(Node root, int key):**
   * Remove um nó da árvore com a chave **key**. O método usa uma abordagem recursiva para encontrar o nó a ser removido. Existem três casos:
     + O nó não tem filhos ou tem apenas um filho: O nó é removido diretamente.
     + O nó tem dois filhos: Encontra o nó sucessor (o menor nó na subárvore direita), substitui o valor do nó a ser removido pelo valor do sucessor e, em seguida, remove o sucessor recursivamente.
   * Após a remoção, atualiza a altura do nó e verifica o fator de equilíbrio. Se necessário, realiza rotações para manter a árvore balanceada.

Essencialmente, os métodos de inserção e remoção garantem que a árvore AVL permaneça balanceada após as operações, realizando rotações quando necessário para manter as propriedades da árvore AVL. Isso ajuda a garantir um tempo de busca eficiente em árvores de busca binária.

Método delete()

O método delete() da árvore AVL remove um nó com a chave dada da árvore mantendo o equilíbrio. Ele funciona da seguinte forma:

1. Procura o nó a ser removido: o método delete() chama recursivamente o método findNode() para encontrar o nó com a chave dada. Se o nó não for encontrado, o método retorna null.
2. Remove o nó: se o nó for encontrado, o método verifica se ele tem um filho ou nenhum filho. Se o nó tiver apenas um filho, o método remove o nó e aponta o pai do nó para o filho remanescente. Se o nó não tiver filhos, o método simplesmente remove o nó.
3. Ajusta a altura e o fator de balanceamento: o método delete() então ajusta a altura e o fator de balanceamento de todos os nós afetados pela remoção.
4. Realiza rotações se necessário: se a árvore ficar desequilibrada após a remoção, o método delete() realiza rotações para restaurar o equilíbrio.

Método insert()

O método insert() da árvore AVL insere um nó com a chave dada na árvore mantendo o equilíbrio. Ele funciona da seguinte forma:

1. Procura o local para inserir o nó: o método insert() chama recursivamente o método findNode() para encontrar o local apropriado para inserir o nó.
2. Insere o nó: se o nó não for encontrado, o método insert() cria um novo nó com a chave dada e o insere no local encontrado.
3. Ajusta a altura e o fator de balanceamento: o método insert() então ajusta a altura e o fator de balanceamento de todos os nós afetados pela inserção.
4. Realiza rotações se necessário: se a árvore ficar desequilibrada após a inserção, o método insert() realiza rotações para restaurar o equilíbrio.

Métodos auxiliares

Os métodos auxiliares toArrayHelper() e minValueNode() são usados pelo método printTree() para imprimir a árvore de forma recursiva. O método toArray() é usado pela função principal para converter a árvore em um array.

Exemplos

Considere a seguinte árvore AVL:

20

/ \

10 30

Para inserir o nó com a chave 15 na árvore, o método insert() faria o seguinte:

1. O método findNode() encontraria o nó com a chave 20.
2. O método insert() criaria um novo nó com a chave 15 e o inseriria no local encontrado.
3. O método insert() ajustaria a altura e o fator de balanceamento do nó 20.
4. O método insert() não realizaria nenhuma rotação, pois a árvore ainda estaria equilibrada.

Após a inserção, a árvore ficaria assim:

20

/ \

10 30

/

15

Para remover o nó com a chave 15 da árvore, o método delete() faria o seguinte:

1. O método findNode() encontraria o nó com a chave 15.
2. O método delete() verificaria se o nó tem um filho ou nenhum filho. Como o nó 15 tem um filho, o método delete() removeria o nó e apontaria o pai do nó para o filho remanescente.
3. O método delete() ajustaria a altura e o fator de balanceamento do nó 20.
4. O método delete() realizaria uma rotação à direita para restaurar o equilíbrio.

Após a remoção, a árvore ficaria assim:

20

/

10

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Método rotateLeft()

O método rotateLeft() realiza uma rotação à esquerda em um nó da árvore AVL. Ele funciona da seguinte forma:

1. Ajusta a altura dos nós envolvidos: o método rotateLeft() ajusta a altura dos nós pai e filho antes de realizar a rotação.
2. Realiza a rotação: o método rotateLeft() troca os ponteiros entre o pai e o filho da esquerda do nó.
3. Ajusta a altura dos nós envolvidos: o método rotateLeft() ajusta a altura dos nós pai e filho novamente após a rotação.

Método rotateRight()

O método rotateRight() realiza uma rotação à direita em um nó da árvore AVL. Ele funciona da seguinte forma:

1. Ajusta a altura dos nós envolvidos: o método rotateRight() ajusta a altura dos nós pai e filho antes de realizar a rotação.
2. Realiza a rotação: o método rotateRight() troca os ponteiros entre o pai e o filho da direita do nó.
3. Ajusta a altura dos nós envolvidos: o método rotateRight() ajusta a altura dos nós pai e filho novamente após a rotação.

Método balanceFactor()

O método balanceFactor() calcula o fator de balanceamento de um nó da árvore AVL. O fator de balanceamento é a diferença entre as alturas das subárvores esquerda e direita do nó. Um fator de balanceamento positivo indica que a subárvore esquerda é mais alta, um fator de balanceamento negativo indica que a subárvore direita é mais alta e um fator de balanceamento nulo indica que as subárvores têm a mesma altura.

Método height()

O método height() calcula a altura de um nó da árvore AVL. A altura de um nó é a distância do nó até a folha mais profunda da árvore.

Método minValueNode()

O método minValueNode() encontra o nó com a chave mínima em uma árvore AVL. Ele funciona da seguinte forma:

1. Começa na raiz da árvore: o método minValueNode() começa na raiz da árvore.
2. Se o nó não tiver filho esquerdo: se o nó não tiver filho esquerdo, o nó é o nó com a chave mínima.
3. Se o nó tiver filho esquerdo: se o nó tiver filho esquerdo, o método minValueNode() chama recursivamente o método minValueNode() no filho esquerdo do nó.

Exemplos

Considere a seguinte árvore AVL:

20

/ \

10 30

O fator de balanceamento do nó 20 é 0, pois as subárvores esquerda e direita têm a mesma altura. O fator de balanceamento do nó 10 é 1, pois a subárvore esquerda é mais alta. O fator de balanceamento do nó 30 é -1, pois a subárvore direita é mais alta.

Para calcular a altura da árvore, o método height() começaria na raiz da árvore. O nó 20 tem altura 1, pois sua subárvore esquerda é vazia. O nó 10 tem altura 2, pois sua subárvore esquerda tem altura 1. O nó 30 tem altura 1, pois sua subárvore esquerda é vazia. Portanto, a altura da árvore é 2.

Para encontrar o nó com a chave mínima, o método minValueNode() começaria na raiz da árvore. O nó 20 não tem filho esquerdo, portanto, ele é o nó com a chave mínima.

Após a rotação à esquerda no nó 20, a árvore ficaria assim:

10

/ \

5 20

O fator de balanceamento do nó 10 é agora 0, pois as subárvores esquerda e direita têm a mesma altura. O fator de balanceamento do nó 20 é 1, pois a subárvore esquerda é mais alta. O fator de balanceamento do nó 5 é -1, pois a subárvore direita é mais alta.

A altura da árvore ainda é 2.

O método minValueNode() ainda retornaria o nó 10, pois ele ainda é o nó com a chave mínima.