Árvore AVL

Uma Árvore AVL é uma árvore de busca binária auto-balanceada, onde a diferença de altura entre as sub-árvores esquerda e direita é mantida como no máximo 1 ou -1. Isso garante que as operações de busca, inserção e remoção tenham tempo de complexidade médio O(log n).

O equilíbrio de uma AVL é medido subtraindo o número de níveis na sub-árvore da esquerda do número de níveis da sub-árvore da direita.

O equilíbrio é corrigido através de rotações, sendo elas:

- Rotação à esquerda
- Rotação à direita
- Rotação dupla à esquerda
- Rotação dupla à direita

É importante ressaltar que o um nó, agora, também mantém o valor de sua altura

```
class Node {
    int key, height;
    Node left, right;

    Node(int d) {
        key = d;
        height = 1;
    }
}
```

Função para descobrir a altura de um nó

```
    int height(Node N) {
        if (N == null)
            return 0;

        return N.height;
    }
```

Função para descobrir o balancemento

```
    int getBalance(Node N) {
        if (N == null)
            return 0;

        return height(N.left) - height(N.right);
    }
```

Função para rotacionar à direita

```
Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    // Perform rotation
    x.right = y;
    y.left = T2;

    // Update heights
    y.height = max(height(y.left), height(y.right)) + 1;
    x.height = max(height(x.left), height(x.right)) + 1;

    // Return new root
    return x;
}
```

Função para rotacionar à esquerda

```
Node leftRotate(Node x) {
    Node y = x.right;
    Node T2 = y.left;

    // Perform rotation
    y.left = x;
    x.right = T2;

    //  Update heights
    x.height = max(height(x.left), height(x.right)) + 1;
    y.height = max(height(y.left), height(y.right)) + 1;

    // Return new root
    return y;
}
```

Função para inserir um nó

```
Node insert(Node node, int key) {

    /* 1.  Perform the normal BST insertion */
    if (node == null)
        return (new Node(key));

    if (key < node.key)
        node.left = insert(node.left, key);
    else if (key > node.key)
```

```
            node.right = insert(node.right, key);
        else // Duplicate keys not allowed
            return node;

        /* 2. Update height of this ancestor node */
        node.height = 1 + max(height(node.left),
                              height(node.right));

        /* 3. Get the balance factor of this ancestor
              node to check whether this node became
              unbalanced */
        int balance = getBalance(node);

        // Rotação à direita
        if (balance > 1 && key < node.left.key)
            return rightRotate(node);

        // Rotação à esquerda
        if (balance < -1 && key > node.right.key)
            return leftRotate(node);

        // Rotação dupla à direita
        if (balance > 1 && key > node.left.key) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }

        // Rotação dupla à esquerda
        if (balance < -1 && key < node.right.key) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        /* return the (unchanged) node pointer */
        return node;
    }
```

Deleção de um nó

```
    Node deleteNode(Node root, int key)
    {
        // STEP 1: PERFORM STANDARD BST DELETE
        if (root == null)
            return root;

        // If the key to be deleted is smaller than
        // the root's key, then it lies in left subtree
        if (key < root.key)
```

```
    root.left = deleteNode(root.left, key);

// If the key to be deleted is greater than the
// root's key, then it lies in right subtree
else if (key > root.key)
    root.right = deleteNode(root.right, key);

// if key is same as root's key, then this is the node
// to be deleted
else
{

    // node with only one child or no child
    if ((root.left == null) || (root.right == null))
    {
        Node temp = null;
        if (temp == root.left)
            temp = root.right;
        else
            temp = root.left;

        // No child case
        if (temp == null)
        {
            temp = root;
            root = null;
        }
        else // One child case
            root = temp; // Copy the contents of
                         // the non-empty child
    }
    else
    {

        // node with two children: Get the inorder
        // successor (smallest in the right subtree)
        Node temp = minValueNode(root.right);

        // Copy the inorder successor's data to this node
        root.key = temp.key;

        // Delete the inorder successor
        root.right = deleteNode(root.right, temp.key);
    }
}

// If the tree had only one node then return
if (root == null)
```

```
            return root;

        // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
        root.height = max(height(root.left), height(root.right)) +
1;

        // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check
whether
        // this node became unbalanced)
        int balance = getBalance(root);

        // If this node becomes unbalanced, then there are 4 cases
        // Left Left Case
        if (balance > 1 && getBalance(root.left) >= 0)
            return rightRotate(root);

        // Left Right Case
        if (balance > 1 && getBalance(root.left) < 0)
        {
            root.left = leftRotate(root.left);
            return rightRotate(root);
        }

        // Right Right Case
        if (balance < -1 && getBalance(root.right) <= 0)
            return leftRotate(root);

        // Right Left Case
        if (balance < -1 && getBalance(root.right) > 0)
        {
            root.right = rightRotate(root.right);
            return leftRotate(root);
        }

        return root;
    }
```