```
#ifndef AVL_TREE_H
#define AVL_TREE_H

#include<iostream>
#include<iomanip>
#include<algorithm>
#include<memory>

template <typename T> //Se usa para usar cualquier tipo de dato basico de
programacion.
class Node { //Aqui es la clase del nodo.
public:
    T data; //Donde se almacena los datos segun el template.
    int height; // Aqui va la altura
    std::shared_ptr<Node<T>> left; //Aqui tan los hijos
    std::shared_ptr<Node<T>> right;

    Node(T data) : data(data), height(1), left(nullptr), right(nullptr) {} // Aqui
es un constructor explicito y resive parametros y los guarda en el nodo.
};

template <typename T> //Cada clase necesita template
class AVLTree { //Aqui empieza la clase
public:
    std::shared_ptr<Node<T>> root; //Es el head del arbol.

    AVLTree(): root(nullptr) {} //Aqui va otro constructor explicito

    void add(T data) { //Aqui se añade los datos al arbol.
        root = insert(root, data);
    }

    void remove(T data)  { //
        root = deleteNode(root, data); //Funcion para borrar el nodo
    }

    void print() { //Aqui es el print, para imprimir la informacion dentro de
void.
        if (root != nullptr) { //Aqui es si root no es igual a null, significa que
si tiene algo.
            print(root, 0);
        } else {
            std::cout << "The tree is empty." << std::endl;
        }
    }

private:
    void print(std::shared_ptr<Node<T>> node, int indent) { //Aqui se pone un
espacio, para saber que tanto tienes que alejarlo del inicio.
        if(node) { //Aqui significa que node existe osea, no es null.
            if(node->right) { //Aqui dice si el nodo es mayor, va hacia la
derecha.
```

```
                print(node->right, indent + 8); //Aqui va aver mas espacio y sigue
estando en la derecha.
            }
            if (indent) { //Aqui es de que indent existe, no es null.
                std::cout << std::setw(indent) << ' '; //Aqui es un vacio para
acomodar la jerarquia.
            }
            if (node->right) { // Si el nodo tiene la derecha, aqui se imprime el
hijo de el nodo de la derecha
                std::cout << " / (Right of " << node->data << ")\n" <<
std::setw(indent) << ' ';
            }
            std::cout << node->data << "\n" ;
            if (node->left) { //Aqui ahora se imprime la izquierda.
                std::cout << std::setw(indent) << ' ' << " \\ (Left of " << node-
>data << ")\n";
                print(node->left, indent + 8);
            }
        }
    }

    std::shared_ptr<Node<T>> newNode(T data) { //Aqui es un nuevo nodo con el t
data.
        return std::make_shared<Node<T>>(data); //Aqui solo se hace un nodo con
los datos arriba.
    }

    std::shared_ptr<Node<T>> rightRotate(std::shared_ptr<Node<T>> y) { //Rota al
nodo de la derecha.
        std::shared_ptr<Node<T>> x = y->left; //Asigna un valor x hacia el hijo
izquierdo de y.
        std::shared_ptr<Node<T>> T2 = x->right; // Asigna el valor T2 al valor del
hijo derecho de x.

        x->right = y;
        y->left = T2;

        y->height = max(height(y->left), height(y->right))+1; //Aqui se calcula la
altura de x y y.
        x->height = max(height(x->left), height(x->right))+1;

        return x;
    }

    std::shared_ptr<Node<T>> leftRotate(std::shared_ptr<Node<T>> x) {
        std::shared_ptr<Node<T>> y = x->right; //Aqui se asigna valor y al hijo
derecho de x.
        std::shared_ptr<Node<T>> T2 = y->left; //Aqui se asigna valor T2 al hijo
izquierdo de y.

        y->left = x; //Asigna x al hijo izquiedo de y.
        x->right = T2; //Asigna T2 al hijo izquierdo de x.

        x->height = max(height(x->left),height(x->right))+1; //Y aqui se calcula
```

```
la altura de x Y y.
        y->height = max(height(y->left),height(y->right))+1;

        return y;
    }

    int getBalance(std::shared_ptr<Node<T>> N) {
        if (N == nullptr) // Si el null es cero, esta balanceado.
            return 0;
        return height(N->left) - height(N->right); //Si el nodo no es null,
extracta la altura de la derecha y izquierda.
    }

    std::shared_ptr<Node<T>> insert(std::shared_ptr<Node<T>> node, T data) {
//Aqui empieza la funcion.
        if (node == nullptr) //Aqui es donde si no hay nada, se guarda algo.
            return (newNode(data)); //Aqui lo guarda.

        if (data < node->data) //Aqui es si la informacion es menor que data
            node->left = insert(node->left, data); //Le pasamos la informacion de
la izquiera y que se mantega ahi.
        else if (data > node->data) //Si fuera mayor, se va a la derecha.
            node->right = insert(node->right, data); //Aqui es lo mismo que cuando
la informacion es en la izquiera.
        else
            return node; //Aqui solo regresamo el nuemero que ya tiene en caso de
que el numero sea igual.

        node->height = 1 + max(height(node->left), height //Calcula la altura
despues de la inserction.  (node->right)); //

        int balance = getBalance(node); //Aqui se crea un int para checar el
balance usando la extraccion previamente definida.

        if (balance > 1 && data < node->left->data) //Si el balance es mas que 1 y
si los data es menor que los datos del hijo izquierdo del nodo.
            return rightRotate(node); //Rota de manera a la par de manesillas de
un reloj de dicho nodo.

        if (balance < -1 && data > node->right->data) //Aqui, si el balance es
menor que -1 y data es mayor que data en el hijo derecho del nodo.
            return leftRotate(node);

        if (balance > 1 && data > node->left->data) { //Aqui es cuando tanto
balance, como data es que son mayores que el hijo izquierdo del nodo.
            node->left = leftRotate(node->left); //Gira en sentido antihorario en
el hijo izquierdo del nodo y asigna un nuevo valor de rotación al nodo
            return rightRotate(node);
        }

        if (balance < -1 && data < node->right->data) { //Igual que arriba, si
balance y data son menores que -1 data del hijo derecho del nodo.
            node->right = rightRotate(node->right); //Gira en sentido antihorario
en el hijo derecho del nodo y asigna un nuevo valor de rotación al nodo.
```

```
            return leftRotate(node);
        }

        return node;
    }

    std::shared_ptr<Node<T>> minValueNode(std::shared_ptr<Node<T>> node) {
        std::shared_ptr<Node<T>> current = node; //Declara un nuevo nodo llamado
actual con el valor del nodo actual

        while (current->left != nullptr) //Mientras el hijo izquierdo del nodo
actual no sea nulo
            current = current->left; //Declara el nodo actual como su hijo
izquierdo.

        return current;
    }

    std::shared_ptr<Node<T>> deleteNode(std::shared_ptr<Node<T>> root, T data) {
        if (!root) //Aqui es de que si el arbol esta vacio, no hay nada que
guardar.
            return root;

        if (data < root->data) { // Como data es menor que root, se busca en la
izquierda.
            root->left = deleteNode(root->left, data); //Sigue buscando por la
izquierda.
        }
        else if(data > root->data) {
            root->right = deleteNode(root->right, data); //Aqui sigue buscando por
la derecha.
        }
        else {
            if(!root->left || !root->right) { //Aqui comprueba si el hijo
izquierdo o derecho de la raíz está vacío
                root = (root->left) ? root->left : root->right; //Aqui si existe
el hijo izquierdo de la raíz, se asigna al hijo izquierdo de la raíz. Si no,
asigna el hijo derecho de la raíz.
            }
            else {
                std::shared_ptr<Node<T>> temp = minValueNode(root->right);
//Asigna al nodo temporal el valor mínimo del lado derecho de la raíz
                root->data = temp->data; //Asigna data hacia root el data de temp.
                root->right = deleteNode(root->right, temp->data); //Y aqui se
declara del lado derecho del nodo
                temp.reset();
            }
        }

        if(!root) //Si root existe
            return root;

        root->height = 1 + max(height(root->left), height(root->right)); //Aqui
calcula la altura despues de la surpresion.
```

```
        int balance = getBalance(root); //Aqui crea int para el equilibrio desde
la root.

        if (balance > 1 && getBalance(root->left) >= 0) //Si balance es mayor que
1 Y el balance del hijo izquierdo de root es mayor o igual a 0
            return rightRotate(root); //Aqui se rota a base de la manesillas del
reloj en root.

        if (balance < -1 && getBalance(root->right) <= 0) //Si balance es menor
que -1 Y el balance del hijo derecho de root es menor o igual a 0
            return leftRotate(root); //Aqui se rota de manera opuesta a la
manesillas del reloj en root.

        if (balance > 1 && getBalance(root->left) < 0) { //Si balance es menor que
-1 Y el balance del hijo derecho de root es mayor que 0
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }

        if (balance < -1 && getBalance(root->right) > 0) { //Si balance es menor
que -1 Y balance del hijo derecho de root es mayor que 0
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }

        return root;
    }

    int height(std::shared_ptr<Node<T>> N) {
        if (N == nullptr) //Checha si N es nulo, si es asi, retorna 0
            return 0;
        return N->height; //Si no, retorna la altura de N
    }

    int max(int a, int b) {
        return (a > b)? a : b; //Y aqui checa si a es mayor a b, si es asi,
retorna a, si no, b.
    }
};

#endif /* AVL_TREE_H */
```