Documentacion de un arbol

Arbol ALV

es un árbol binario de búsqueda (ABB) en el que las alturas de los dos subárboles de cada nodo difieren a lo más en 1. El balance de un nodo en un árbol binario en general y de un árbol AVL en particular, se define como la altura de su subárbol izquierdo menos la altura de su subárbol derecho: | altura(arbolIzquierdo) - altura(arbolDerecho) | < 2

Rotaciones

es una modificación sencilla sobre la estructura de un árbol binario de búsqueda, que permite mantener la propiedad de orden.

Rotacion Simple: Rotación simple izquierda es cuando un nodo está cargado a la derecha, pero no está cargado a la izquierda. Rotación simple derecha es cuando un nodo está cargado a la izquierda, pero no está cargado a la derecha. Rotacion Doble: es un proceso un poco más elaborado que el de la rotación simple, ya que como su nombre lo indica, implica dos rotaciones.

nullptr: estamos apuntando a un nodo que no tiene un valor. **Shared_ptr**: permite que apunte al mismo valor sin consecuencias

```
#ifndef AVL_TREE_H
#define AVL_TREE_H

#include<iostream>
#include<iomanip> //Extrae un valor monetario de un flujo con el formato
especificado y devuelve el valor en un parámetro.
#include<algorithm>
#include<memory> // obtiene la direccion real de un objeto

template <typename T>
class Node { // asignamos los nodos
public:
    T data; // informacion que guarda el nodo
    int height;
    std::shared_ptr<Node<T>> left; // asignamos el nodo izquierdo
    std::shared_ptr<Node<T>> right; // asignamos el nodo derecho

// constructor implicito
    Node(T data) : data(data), height(1), left(nullptr), right(nullptr) {} //
Recibe un parametro, data que se guarda en la variable data
};

template <typename T> // asiganmos el template
class AVLTree { //creamos la clase arbol alv(balaceado o equilibrado)
public:
    std::shared_ptr<Node<T>> root; //esta compartiendo las particularidades del
nodo "T" a la raiz

// constructor implicito, la raiz se guarda como nulo
```

```
    AVLTree(): root(nullptr) {}

    void add(T data) { // agregar el nodo "T"
        root = insert(root, data);
    }

    void remove(T data)  { // eliminar el nodo "T"
        root = deleteNode(root, data);
    }

    void print() { // imprime el resultado de si la raiz se detecta como nulo, se
dice que el arbol esta vacio si no se imprime el valor de la raiz
        if (root != nullptr) {
            print(root, 0);
        } else {
            std::cout << "The tree is empty." << std::endl;
        }
    }

private:
    void print(std::shared_ptr<Node<T>> node,  int indent) {
        if(node) {  // imprime la profundidad del nodo a 8, considerando que
inicia en 0, primero empieza con la derecha y despues hace el mismo procedimiento
con la izquierda
            if(node->right) { // sigue el caminodel arbol hacia la derecha
                print(node->right, indent + 8);
            }
            if (indent) {
                std::cout << std::setw(indent) << ' ';
            }
            if (node->right) { // imprime el camino a la derecha del arbol
                std::cout << " / (Right of " << node->data << ")\n" <<
std::setw(indent) << ' ';
            }
            std::cout << node->data << "\n" ;
            if (node->left) { // imprime el camino a la izquiersa del arbol
                std::cout << std::setw(indent) << ' ' << " \\ (Left of " << node-
>data << ")\n";
                print(node->left, indent + 8);
            }
        }
    }

    std::shared_ptr<Node<T>> newNode(T data) {
        return std::make_shared<Node<T>>(data);
    }

    std::shared_ptr<Node<T>> rightRotate(std::shared_ptr<Node<T>> y) { // hace la
rotacion derecha para equilibrarlo
        std::shared_ptr<Node<T>> x = y->left; // se guarda el hijo izquierdo con
el nombre "x"
        std::shared_ptr<Node<T>> T2 = x->right; // se guarda el hijo derecho de
"x" con el nombre "T2"
```

```
        x->right = y; // hacemos que el hijo derecho de "x" sea "y"
        y->left = T2; // hacemos que el hijo izquierdo de "y" sea "T2"

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

        return x;
    }

    std::shared_ptr<Node<T>> leftRotate(std::shared_ptr<Node<T>> x) { // hace la
rotacion izquierdo para equilibrarlo
        std::shared_ptr<Node<T>> y = x->right; // se guarda el hijo derecho con el
nombre "y"
        std::shared_ptr<Node<T>> T2 = y->left; // se guarda el hijo derecho de "y"
con el nombre "T2"

        y->left = x; // hacemos que el hijo izquierdo de "y" sea "x"
        x->right = T2; // hacemos que el hijo derecho de "x" sea "T2"

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

        return y;
    }

    int getBalance(std::shared_ptr<Node<T>> N) {
        if (N == nullptr)// si el nodo es hoja su valor es 0
            return 0;
        return height(N->left) - height(N->right); // si no es hoja, se obtiene
con la altura del sub-arbol izquierdo menos la altura del sub-arbol derecho
    }

    std::shared_ptr<Node<T>> insert(std::shared_ptr<Node<T>> node, T data) {
        if (node == nullptr)// una vez que llega a un puntero vacio o nulo, crea
un nuevo nodo con informacion pasada
            return (newNode(data));

        if (data < node->data) // si el nodo no es nulo y la informacion es menor
al nodo, se continua la busqueda por el nodo izquierdo
            node->left = insert(node->left, data);
        else if (data > node->data) // si el nodo no es nulo y sla informacion es
mayor al nodo, se continua la busqueda por el nodo derecho
            node->right = insert(node->right, data);
        else // en caso de que ya exista un nodo con esa informacion, se utiliza
ese
            return node;

        node->height = 1 + max(height(node->left), height(node->right));

        int balance = getBalance(node); // se consigue el factor equilibrio del
```

```
nuevo nodo

        if (balance > 1 && data < node->left->data) // caso de desbalance
izquierdo-izquierdo
            return rightRotate(node);

        if (balance < -1 && data > node->right->data) // caso de desbalance
derecho-derecho
            return leftRotate(node);

        if (balance > 1 && data > node->left->data) { // caso de desbalance
izquierdo-derecho
            node->left = leftRotate(node->left); // se realiza una rotacion
izquierda con el hijo izquierdo
            return rightRotate(node);
        }

        if (balance < -1 && data < node->right->data) { // caso de desbalance
derechio-izquierdo
            node->right = rightRotate(node->right); // se realiza una rotacion
derecha con el hijo derecho
            return leftRotate(node);
        }

        return node;
    }

    std::shared_ptr<Node<T>> minValueNode(std::shared_ptr<Node<T>> node) {
        std::shared_ptr<Node<T>> current = node;

        // se avanza continuamente por el nodo izquierdo hasta llegar a la hoja
mas izquierda
        while (current->left != nullptr)
            current = current->left;

        return current;
    }

    std::shared_ptr<Node<T>> deleteNode(std::shared_ptr<Node<T>> root, T data) {
        if (!root) // si no encuentra raiz, dejar de buscar
            return root;

        if (data < root->data) { // busca el nodo que se debe borrar por el sub-
arbol izquierdo
            root->left = deleteNode(root->left, data);
        }
        else if(data > root->data) { // busca el nodo que se debe borrar por el
sub-arbol derecho
            root->right = deleteNode(root->right, data);
        }
        else {
            if(!root->left || !root->right) {  // si solo hay un hijo, ese mismo
toma el lugar o remplaza al padre
                root = (root->left) ? root->left : root->right;
```

```
            }
            else {
                std::shared_ptr<Node<T>> temp = minValueNode(root->right);
                root->data = temp->data;
                root->right = deleteNode(root->right, temp->data);
                temp.reset();
            }
        }

        if(!root)
            return root;

        //sigue el mismo procedimiento ya explicado
        root->height = 1 + max(height(root->left), height(root->right));

        int balance = getBalance(root);

        if (balance > 1 && getBalance(root->left) >= 0)
            return rightRotate(root);

        if (balance < -1 && getBalance(root->right) <= 0)
            return leftRotate(root);

        if (balance > 1 && getBalance(root->left) < 0) {
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }

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

        return root;
    }

    int height(std::shared_ptr<Node<T>> N) {
        if (N == nullptr)
            return 0;
        return N->height;
    }

    int max(int a, int b) {
        return (a > b)? a : b;
    }
};

#endif /* AVL_TREE_H */
```