AVLTree.h

```
struct Node {
    int value;
    Node* left;
    Node* right;
    int height;
};
```

Lo que hace esta parte de codigo es un struct en donde se crea el entero "Value" dos nodos punteros uno para la izquierda y el otro para la derecha y por ultimo un entero que sera el tamaño.

```
public:
    void printTree() {
        printTree(root, 0, 10);
    }
```

Esta parte se encargara de imprimir el arbol desde el nodo hasta con un tamaño de 10.

```
int getNodeHeight(Node* N) {
        if (N == nullptr)
            return 0;
        return N->height;
    }
```

Se encarga de conseguir el tamaño de un nodo, en caso de ser 0 este lo volvera hacer, una vez listo convertida height al valor que salio en N.

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

En este caso de encarga de checar cual es el valor mayor comparandolos, si es a regresa un a y viceversa con b.

```
Node* newNode(int value) {
        Node* node = new Node();
        node->value = value;
        node->left = nullptr;
        node->right = nullptr;
        node->height = 1;
        return(node);
    }
```

Es un puntero de nodo en donde primero abre espacio para un nuevo nodo y le asigna una posicion al puntero node, luego se le proceden a dar valores al nodo como value, vacio para el lado izquierdo y derecho y un valor de 1 en height y ya por ultimo se le regresa el puntero al nodo recin creado.

```
Node* rightRotate(Node* y) {
        Node* x = y->left;
        Node* T2 = x->right;

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

        y->height = maxHeight(getNodeHeight(y->left), getNodeHeight(y->right)) + 1;
        x->height = maxHeight(getNodeHeight(x->left), getNodeHeight(x->right)) + 1;

        return x;
    }
```

Se encarga de balancear el arbol en caso de que se le agregen nuevos nodos de modo de que a los child de izquierda y derecha se le da un nodo el cual asumen que tambien tienen un child de igual manera el child de la neresa es asumido como el nuevo root ya por ultimo se vuelve a calcular el height del arbol para ver si cambio o no.

```
Node* leftRotate(Node* x) {
        Node* y = x->right;
        Node* T2 = y->left;

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

        x->height = maxHeight(getNodeHeight(x->left), getNodeHeight(x->right)) + 1;
        y->height = maxHeight(getNodeHeight(y->left), getNodeHeight(y->right)) + 1;

        return y;
    }
```

lo mismo que el anterios solo que esta vez el de la izquierda es el que se convierte en root del arbol.

```
int getBalance(Node* N) {
        if (N == nullptr)
            return 0;
        return getNodeHeight(N->left) - getNodeHeight(N->right);
    }
```

Se encarga de checar el balance del arbol de izquierda y derecha, si N esta vacio se regresa un cero.

```
Node* insertNode(Node* node, int value) {
        if (node == nullptr)
            return(newNode(value));

        if (value < node->value)
            node->left = insertNode(node->left, value);
        else if (value > node->value)
```

```
            node->right = insertNode(node->right, value);
        else
            return node;

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

        int balance = getBalance(node);

        if (balance > 1 && value < node->left->value)
            return rightRotate(node);

        if (balance < -1 && value > node->right->value)
            return leftRotate(node);

        if (balance > 1 && value > node->left->value) {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }

        if (balance < -1 && value < node->right->value) {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }

        return node;
    }
```

Se encarga de insertarle un nodo al arbol, primero si el nodo actual esta vacio se regresa el nuevo nodo creado con anterioridad, en caso de que el valor del nodo insertado sea menor al valor del nodo actual es enviado a la izquierda de lo contrario es enviado a la derecha para despues agregarle un mas 1 a la altura de ambos nodos izq y der, ya por ultimo se crea un entero que cheque por el balance del arbol, si el balance es mayor a 1 y el valor del nodo izquierdo es menor se utiliza la funcion de rotar hacia la derecha y viceversa, caso contrario en donde el balance es -1 y el valor del nodo es mayor al nodo derecho se utiliza la funcion de rotar hacia la izquierda y viceversa.

```
Node* minValueNode(Node* node) {
        Node* current = node;

        while (current->left != nullptr)
            current = current->left;

        return current;
    }
```

Se encarga de hacer un loop por el lado izquierdo del arbol desde el nodo para encontrar el menor valor posible dentro de esta, una vez que el loop termina, current apuntara al nodo que le menor valor del arbol.

```
Node* deleteNode(Node* root, int value) {
        if (root == nullptr)
            return root;

        if ( value < root->value )
            root->left = deleteNode(root->left, value);
        else if( value > root->value )
            root->right = deleteNode(root->right, value);
```

```
        else {
            if( (root->left == nullptr) || (root->right == nullptr) ) {
                Node *temp = root->left ? root->left : root->right;

                if(temp == nullptr) {
                    temp = root;
                    root = nullptr;
                }
                else
                    *root = *temp;

                delete temp;
            }
            else {
                Node* temp = minValueNode(root->right);

                root->value = temp->value;

                root->right = deleteNode(root->right, temp->value);
            }
        }

        if (root == nullptr)
            return root;

        root->height = 1 + maxHeight(getNodeHeight(root->left), getNodeHeight(root->right));

        int balance = getBalance(root);

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

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

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

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

        return root;
    }
```

Similar al metodo de insertar nodos, este se encargara de eliminarlos, en caso de que el nodo este vacio no hara nada si hay un valor a la izquierda o derecha del root estos seran eliminados, en caso de que algun nodo de la izquierda o derecha sea igual al root se checara los child de estos, si tiene uno ajustara los punteros de forma apropiada mientras que si tiene dos bolvera el nodo de menor velor del lado derecho el nodo actual para luego eliminar al sucesor.

Ya por ultimo se encargara de ajustar el arbol con los valores restantes usando la misma funcion que se utilizo en el apartado de insertar.

```
void deleteTree(Node* node) {
        if (node == nullptr)
            return;

        deleteTree(node->left);
        deleteTree(node->right);

        delete node;
    }
```

Se encarga de eliminar el arbol primero checando si el nodo esta vacio para luego elimianar los nodos que se encuentren a la derecha o izquierda de este para al final simplemente eliminar el nodo restante.

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

    ~AVLTree() {
        deleteTree(root);
    }

    void insert(int value) {
        root = insertNode(root, value);
    }

    void remove(int value) {
        root = deleteNode(root, value);
    }
```

Aparte de inicializar el arbol AVL primero manda a llamar la funcion deletetree para eliminar cualquier arbol que se encuentre (basicamente empezar desde cero) luego crea dos void uno para insertar nodos y el otro para eliminarlos teniendd el valor y root en ellos.

```
void inorderTraversal(Node* node) {
        if (node == nullptr)
            return;

        inorderTraversal(node->left);
        std::cout << node->value << " ";
        inorderTraversal(node->right);
    }
```

Se encarga de visitar nodos atravez del arbol empezando por la izquierda, luego a la raiz y finalmente a la derecha, todo eso mientras va imprimiendo los valores, en caso de que el nodo este vacio no hara nada.

```
void preorderTraversal(Node* node) {
        if (node == nullptr)
            return;

        std::cout << node->value << " ";
        preorderTraversal(node->left);
        preorderTraversal(node->right);
```

```
    }
```

Al igual que la anterior se encarga de visitar nodos atravez del arbol esta vez empezando por el nodo actual, luego al subarbol de la izquierda para luego pasar al de la derecha mientras va imprimiendo valores.

```
void postorderTraversal(Node* node) {
        if (node == nullptr)
            return;

        postorderTraversal(node->left);
        postorderTraversal(node->right);
        std::cout << node->value << " ";
    }
```

Es igual que los dos anteriores pero esta vez el orden es izquierda, derecha y nodo actual

```
void printTree(Node* root, int space = 0, int COUNT = 10) {
        if (root == nullptr) {
            return;
        }

        space += COUNT;
        printTree(root->right, space);

        std::cout << std::endl;

        for (int i = COUNT; i < space; i++) {
            std::cout << " ";
        }

        std::cout << root->value << "\n";

        printTree(root->left, space);
    }
```

Por ultimo esta funcion se encarga de imprimir el arbol en la terminal mostrando el arbol completo de forma vertical.