Задача.

Ниже представлен алгоритм удаления вершины, описанный в русской Википедии. Для более детального рассмотрения данного алгоритма можно также посмотреть данную статью. [Статья про AVL.]

```
class AVLTree {
public:
    AVLTree() : root(nullptr) {}
    ~AVLTree(){
        if(root) delete root;
    }
    void insert(int value) {
        root = insert(root, value);
    }
    bool contains(int value) {
        return contains(root, value);
    }
    int height() {
        return height(root);
    }
    void remove(int key) {
        _remove(root, key);
    }
private:
    int height(Node * n) {
        if(n) return n->height;
        return 0;
    }
    Node * insert(Node * n, int value){
        if(!n) return new Node(value);
        if(value < n->value) n->left = insert(n->left, value);
        else n->right = insert(n->right, value);
        return balance(n); // после вставки балансируем вершины
    }
    bool contains(Node * n, int value) {
        if(!n) return false; // если долшли до пустой вершины, значит элемента
нет в дереве
        if(n->value == value) return true; // если сопал с вершиной, то значит
нашли
        if(value < n->value) {
            std::cout << value << " < " << n->value << std::endl;
            return contains(n->left, value); // если меньше - ищем в левом
        }
        else {
            std::cout << value << " > " << n->value << std::endl;
            return contains(n->right, value); // если больше -  в правом
        }
    }
    int delta(Node * n) {
        return height(n->right) - height(n->left); // считаем разницу в высотах
    }
    void restore_height(Node * n) {
        n->height = std::max(height(n->left), height(n->right)) + 1;
    }
    Node * right_rotate(Node * a) { // малое правое вращение
        Node* b = a->left;
        a->left = b->right;
        b->right = a;
```

```
        restore_height(a);
        restore_height(b);
        return b;
    }
    Node * left_rotate(Node * b) { // малое левое вращение
        Node* a = b->right;
        b->right = a->left;
        a->left = b;
        restore_height(b);
        restore_height(a);
        return a;
    }
    Node * balance(Node * p) { // балансировка
        restore_height(p); // восстановим высоту
        if(delta(p) == 2) { // если перевесило правое
            if(delta(p->right) < 0) p->right = right_rotate(p->right); // если
у правого перевешивает левое - вращаем
            return left_rotate(p); // вращаем
        } else if(delta(p) == -2) { // симметричные операции
            if(delta(p->left)>0) p->left = left_rotate(p->left);
            return right_rotate(p);
        }
        return p; // если не потребовалась балансировка, то возвращаем эту же
вершину
    }
    Node * findmin(Node * p) {
        return p->left ? findmin(p->left) : p;
    }
    Node * removemin(Node * p) {
        if( p->left == nullptr) {
            return p->right;
        }
        p->left = removemin(p->left);
        return balance(p);
    }
    Node * _remove(Node * p, int k) {
        if(!p) return 0;
        if(k < p->value ) {
            p->left = _remove(p->left, k);
        }
        else if(k > p->value) {
            p->right = _remove(p->right, k);
        } else {
            Node * q = p->left;
            Node * r = p->right;
            delete p;
            if(!r) return q;
            Node * m = findmin(r);
            m->right = removemin(r);
            m->left = q;
            return balance(m);
        }
        return balance(p);
    }
    Node *root;
};
```