1 Scapegoat tree

```cpp
#pragma once

#include <bits/stdc++.h>

using namespace std;

constexpr double pi = 3.14159265358979323846;
constexpr auto sample_alpha = pi / 4;

template <class T, const double& alpha = sample_alpha>
class scapegoat_tree {
    static_assert(0.5 < alpha && alpha < 1);

    struct node {
        using nodeptr = node*;

        nodeptr l, r;
        T x;

        explicit node(const T& x, nodeptr l = nullptr,
                      nodeptr r = nullptr)
            : x(x), l(l), r(r) {}

        ~node() {
            delete l;
            delete r;
        }
    };

    using nodeptr = typename node::nodeptr;

    nodeptr m_root;
    size_t m_size, m_max_size;

    // checks alpha-height condition of max_size (note that not on
    // size!)
    bool alpha_height(size_t depth) {
        // 1 should be also fine, but 2 is more reliable
        return depth <= log(m_max_size) / -log(alpha) + 2;
    }

    // checks alpha-weight condition of the node
    // note that complexity is linear and it is fine, because we run it
    // on a path where this condition holds (except for root) and it
    // sums as a geometric progression
    static bool alpha_weight(const nodeptr h) {
        assert(h != nullptr);

        return alpha * get_size(h) >=
               min(get_size(h->l), get_size(h->r));
    }
```

```

    // computes size of a subtree in a linear time
    static size_t get_size(nodeptr h) {
        if (h == nullptr) return 0;

        return get_size(h->l) + 1 + get_size(h->r);
    }

    // builds a perfectly balanced tree
    template <class InputIt>
    static nodeptr build(InputIt first, InputIt last) {
        if (first == last) return nullptr;

        auto mid = first + (last - first) / 2;

        return new node(*mid, build(first, mid), build(mid + 1, last));
    }

    // output to vector, nothing complicated
    static void to_vector(nodeptr h, vector<T>& ans) {
        if (h == nullptr) return;

        to_vector(h->l, ans);
        ans.push_back(h->x);
        to_vector(h->r, ans);
    }

    // rebulds a subtree, also in linear time
    // note that h is passed through a reference, so we modify its value
    static void rebuild(nodeptr& h) {
        vector<T> tmp;

        to_vector(h, tmp);
        delete h;
        h = build(tmp.begin(), tmp.end());
    }

    // inserts x into h, if there is no such value
    // note that cur_depth also passed through a reference
    // returns true if some ancestor should be rebalanced and false
    // otherwise the vertex is deep if it fails the alpha-height
    // condition for a max_size, therefore alpha-height for size is also
    // failed and there exists some ancestor that fails the alpha-weight
    // condition
    bool insert(nodeptr& h, const T& x, size_t& cur_depth) {
        if (h == nullptr) {
            h = new node(x);
            m_size++;
            m_max_size = max(m_max_size, m_size);

            return !alpha_height(cur_depth);
        }

```

```
        if (h->x == x) return false;

        cur_depth++;
        auto to_balance = insert(x < h->x ? h->l : h->r, x, cur_depth);
        cur_depth--;

        if (to_balance && !alpha_weight(h)) {
            // this rebuild is amortized by the quantity of erase
            // operations performed on the smallest subtree and quantity
            // of insert operations to the largest subtree
            rebuild(h);
            to_balance = false;
        }

        return to_balance;
    }

    // erases the leftmost node of the tree and returns it (so it is not
    // deleted) we need it in the erase_root method
    static nodeptr pop_front(nodeptr& h) {
        assert(h != nullptr);

        if (h->l == nullptr) {
            auto ret = h;
            h = h->r;

            return ret;
        } else
            return pop_front(h->l);
    }

    // erases root of the subtree, new root is the leftmost node of the
    // right subtree runs in time of the height of the tree (i.e.
    // logarithmic) and makes O(1) changes in tree structure
    static void erase_root(nodeptr& h) {
        assert(h != nullptr);

        auto le = h->l;
        auto ri = h->r;
        h->l = h->r = nullptr;

        delete h;

        if (ri == nullptr) {
            h = le;

            return;
        }

        h = pop_front(ri);
        h->l = le;
        h->r = ri;
    }
```

```
    // searches for x and erases it, nothing complicated
    void erase(nodeptr& h, const T& x) {
        if (h == nullptr) return;

        if (h->x == x) {
            erase_root(h);
            m_size--;

            return;
        }

        erase(x < h->x ? h->l : h->r, x);
    }

    // find method, also nothing complicated
    static bool find(nodeptr h, const T& x) {
        if (h == nullptr) return false;
        if (h->x == x) return true;

        return find(x < h->x ? h->l : h->r, x);
    }

   public:
    scapegoat_tree() : m_root(nullptr), m_size(0), m_max_size(0) {}

    template <class InputIt>
    scapegoat_tree(InputIt first, InputIt last)
        : m_root(build(first, last)),
          m_size(distance(first, last)),
          m_max_size(distance(first, last)) {}

    ~scapegoat_tree() { delete m_root; }

    void insert(const T& x) {
        size_t d = 0;

        auto to_balance = insert(m_root, x, d);

        assert(!to_balance);
    }

    void erase(const T& x) {
        erase(m_root, x);

        if (m_size < alpha * m_max_size) {
            // this rebuild is amortized by all erase operations
            rebuild(m_root);
            m_max_size = m_size;
        }
    }

    size_t count(const T& x) const { return find(m_root, x) ? 1 : 0; }
```

```
    [[nodiscard]] size_t size() const { return m_size; }

    vector<T> to_vector() const {
        vector<T> ret;
        ret.reserve(m_size);

        to_vector(m_root, ret);

        return ret;
    }
};
```