数据结构

树状数组

```
template<typename T>
struct Fenwick {
    int n;
    std::vector <T> a;

    Fenwick(int n_ = 0) {
        init(n_);
    }

    void init(int n_) {
        n = n_;
        a.assign(n + 1, T{});
    }

    void add(int x, const T &v) {
        if (x <= 0 || x > n) return;
        for (int i = x; i <= n; i += i & -i) {
            a[i] = a[i] + v;
        }
    }

    T Query(int x) {
        if (x <= 0) return T{};
        if (x > n) x = n;
        T ans{};
        for (int i = x; i != 0; i -= i & -i) {
            ans = ans + a[i];
        }
        return ans;
    }

    T range_Query(int l, int r) {
        if (l > r) return 0;
        return Query(r) - Query(l - 1);
    }

    int kth(const T &k) {
        int x = 0;
        T cur{};
        for (int i = 1 << std::__lg(n); i; i /= 2) {
            if (x + i <= n && cur + a[x + i] < k) {
                x += i;
                cur = cur + a[x];
            }
        }
        return x + 1;
    }
};
```

RMQ

```
/**
 * author:jiangly
 * pretreatment:O(n)
 * Inquire:O(1)
*/
template<class T,
    class Cmp = std::less<T>>
struct RMQ {
    const Cmp cmp = Cmp();
    static constexpr unsigned B = 64;
    using u64 = unsigned long long;
    int n;
    std::vector<std::vector<T>> a;
    std::vector<T> pre, suf, ini;
    std::vector<u64> stk;
    RMQ() {}
    RMQ(const std::vector<T> &v) {
        init(v);
    }
    void init(const std::vector<T> &v) {
        n = v.size();
        pre = suf = ini = v;
        stk.resize(n);
        if (!n) {
            return;
        }
        const int M = (n - 1) / B + 1;
        const int lg = std::__lg(M);
        a.assign(lg + 1, std::vector<T>(M));
        for (int i = 0; i < M; i++) {
            a[0][i] = v[i * B];
            for (int j = 1; j < B && i * B + j < n; j++) {
                a[0][i] = std::min(a[0][i], v[i * B + j], cmp);
            }
        }
        for (int i = 1; i < n; i++) {
            if (i % B) {
                pre[i] = std::min(pre[i], pre[i - 1], cmp);
            }
        }
        for (int i = n - 2; i >= 0; i--) {
            if (i % B != B - 1) {
                suf[i] = std::min(suf[i], suf[i + 1], cmp);
            }
        }
        for (int j = 0; j < lg; j++) {
            for (int i = 0; i + (2 << j) <= M; i++) {
                a[j + 1][i] = std::min(a[j][i], a[j][i + (1 << j)], cmp);
            }
        }
        for (int i = 0; i < M; i++) {
            const int l = i * B;
            const int r = std::min(1U * n, l + B);
            u64 s = 0;
            for (int j = l; j < r; j++) {
                while (s && cmp(v[j], v[std::__lg(s) + l])) {
                    s ^= 1ULL << std::__lg(s);
                }
                s |= 1ULL << (j - l);
                stk[j] = s;
            }
        }
    }
    T operator()(int l, int r) {
        if (l / B != (r - 1) / B) {
            T ans = std::min(suf[l], pre[r - 1], cmp);
            l = l / B + 1;
            r = r / B;
            if (l < r) {
                int k = std::__lg(r - l);
                ans = std::min({ans, a[k][l], a[k][r - (1 << k)]}, cmp);
            }
            return ans;
        } else {
            int x = B * (l / B);
            return ini[__builtin_ctzll(stk[r - 1] >> (l - x)) + l];
        }
    }
};
```

线段树

单点

```
template<class Info>
struct SegmentTree {
    int n;
    std::vector<Info> info;
    SegmentTree() : n(0) {}
    SegmentTree(int n_, Info v_ = Info()) {
        init(n_, v_);
    }
    template<class T>
    SegmentTree(std::vector<T> init_) {
        init(init_);
    }
    void init(int n_, Info v_ = Info()) {
        init(std::vector(n_, v_));
    }
    template<class T>
    void init(std::vector<T> init_) {
        n = init_.size();
        info.assign(4 << std::__lg(n), Info());
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                info[p] = init_[l];
                return;
            }
            int m = (l + r) / 2;
            build(2 * p, l, m);
            build(2 * p + 1, m, r);
            pull(p);
        };
        build(1, 0, n);
    }
    void pull(int p) {
        info[p] = info[2 * p] + info[2 * p + 1];
    }
    void modify(int p, int l, int r, int x, const Info &v) {
        if (r - l == 1) {
            info[p] = v;
            return;
        }
        int m = (l + r) / 2;
        if (x < m) {
            modify(2 * p, l, m, x, v);
        } else {
            modify(2 * p + 1, m, r, x, v);
        }
        pull(p);
    }
    void modify(int p, const Info &v) {
        modify(1, 0, n, p, v);
    }
    Info rangeQuery(int p, int l, int r, int x, int y) {
        if (l >= y || r <= x) {
            return Info();
        }
        if (l >= x && r <= y) {
            return info[p];
        }
        int m = (l + r) / 2;
        return rangeQuery(2 * p, l, m, x, y) + rangeQuery(2 * p + 1, m, r, x, y);
    }
    Info rangeQuery(int l, int r) {
        return rangeQuery(1, 0, n, l, r);
    }
    template<class F>
    int findFirst(int p, int l, int r, int x, int y, F pred) {
        if (l >= y || r <= x || !pred(info[p])) {
            return -1;
        }
        if (r - l == 1) {
            return l;
        }
        int m = (l + r) / 2;
        int res = findFirst(2 * p, l, m, x, y, pred);
        if (res == -1) {
            res = findFirst(2 * p + 1, m, r, x, y, pred);
        }
        return res;
    }
    template<class F>
    int findFirst(int l, int r, F pred) {
        return findFirst(1, 0, n, l, r, pred);
    }
    template<class F>
    int findLast(int p, int l, int r, int x, int y, F pred) {
        if (l >= y || r <= x || !pred(info[p])) {
            return -1;
        }
        if (r - l == 1) {
            return l;
        }
        int m = (l + r) / 2;
        int res = findLast(2 * p + 1, m, r, x, y, pred);
        if (res == -1) {
            res = findLast(2 * p, l, m, x, y, pred);
        }
        return res;
    }
    template<class F>
    int findLast(int l, int r, F pred) {
        return findLast(1, 0, n, l, r, pred);
    }
};

struct Info {
    int x = 0;
    int cnt = 0;
};

Info operator+(Info a, Info b) {
    if (a.x == b.x) {
        return {a.x, a.cnt + b.cnt};
    } else if (a.cnt > b.cnt) {
        return {a.x, a.cnt - b.cnt};
    } else {
        return {b.x, b.cnt - a.cnt};
    }
}
```

区间

```
template<class Tag, class Info>
struct LazySegmenttree {
    int n;
    std::vector <Info> info;
    std::vector <Tag> tag;

    LazySegmenttree() : n(0) {}

    LazySegmenttree(const int &n, const Info &x = Info()) {
        init(n, x);
    }
    template<class T>
    LazySegmenttree(const std::vector <T> &v) {
        init(v);
    }

    void init(int n, const Info &x = Info()) {
        init(std::vector<Info>(n, x));
    }

    template<class T>
    void init(const std::vector <T> &v) {
        n = (int) v.size();
        info.assign(4 << std::__lg(n), Info());
        tag.assign(4 << std::__lg(n), Tag());
        std::function<void(int, int, int)>
        build = [&](int l, int r, int p) {
            if ((r - l) == 1) {
                info[p] = v[l];
                return;
            }
            int mid = (r + l) >> 1;
            build(l, mid, p << 1);
            build(mid, r, p << 1 | 1);
            pull(p);
        };
        build(0, n, 1);
    }

    void apply(int p, const Tag &x) {
        info[p].apply(x);
        tag[p].apply(x);
    }

    void push(int p) {
        apply(p << 1, tag[p]);
        apply(p << 1 | 1, tag[p]);
        tag[p] = Tag();
    }

    void pull(int p) {
        info[p] = info[p << 1] + info[p << 1 | 1];
    }

    void range_Change(int x, int y, const Tag &tag) {
        std::function<void(int, int, int)>
        range_Change = [&](int l, int r, int p) {
            if (y <= l || r <= x) return;
            if (x <= l && r <= y) {
                apply(p, tag);
                return;
            }
            int mid = (l + r) >> 1;
            push(p);
            range_Change(l, mid, p << 1);
            range_Change(mid, r, p << 1 | 1);
            pull(p);
        };
        range_Change(0, n, 1);
    }

    Info range_Query(int x, int y) {
        std::function<Info(int, int, int)>
        range_query = [&](int l, int r, int p) {
            if (y <= l || r <= x) {
                return Info();
            }
            if (x <= l && r <= y) {
                return info[p];
            }
            int mid = (l + r) >> 1;
            push(p);
            return range_query(l, mid, p << 1)
                   + range_query(mid, r, p << 1 | 1);
        };
        return range_query(0, n, 1);
    }

    void show(int x, int y) {
        std::function<void(int, int, int)>
        show = [&](int l, int r, int p) {
            if (y <= l || r <= x) {
                return;
            }
            if (r - l == 1) {
                info[p].show();
                return;
            }
            int mid = (l + r) >> 1;
            push(p);
            show(l, mid, p << 1);
            show(mid, r, p << 1 | 1);
        };
        show(0, n, 1);
        cerr << endl;
    }

    void show() {
        show(0, n);
    }
};

struct Tag {
    i64 add = 0;

    void apply(const Tag &x) &{
        add += x.add;
    }
};

struct Info {
    i64 val = 0, l = 1;
    void apply(const Tag &x) &{
        val += x.add * l;
    }

    void show() {
        cerr << val << ' ';
    }
};

Info operator+(const Info &a, const Info &b) {
    return {a.val + b.val, a.l + b.l};
}

using SegmentTree =
    LazySegmenttree<Tag, Info>;
```