LCA

RMQLCA

```
struct edge { int v, w; };
vector<edge> g[N];
int et[N << 1], ei[N], dep[N], ec; ll dis[N];
void dfs_lca(int u, int f, ll w) {
    et[++ec] = u; ei[u] = ec;
    dep[u] = dep[f] + 1; dis[u] = w;
    for (edge e : g[u]) if (e.v != f)
        dfs_lca(e.v, u, w + e.w), et[++ec] = u;
}

int lg2[N << 1], st[19][N << 1];
void build_lca() {
    for (int i = 2; i <= ec; ++i) lg2[i] = lg2[i >> 1] + 1;
    for (int i = 1; i <= ec; ++i) st[0][i] = et[i];
    for (int j = 1; (1 << j) <= ec; ++j) {
        for (int i = 1; i + (1 << j) - 1 <= ec; ++i) {
            int u = st[j - 1][i], v = st[j - 1][i + (1 << j - 1)];
            st[j][i] = dep[u] < dep[v] ? u : v;
        }
    }
}

int lca(int u, int v) {
    int l = ei[u], r = ei[v];
    if (l > r) swap(l, r);
    ++r; int w = lg2[r - l];
    u = st[w][l]; v = st[w][r - (1 << w)];
    return dep[u] < dep[v] ? u : v;
}

ll qdis(int u, int v) {
    return dis[u] + dis[v] - 2 * dis[lca(u, v)];
}
```

```
//  Example:
ec = 0; dfs_lca(1, 0, 0);
build_lca();
```

Tarjan LCA

树剖

重链剖分

将树上路径操作转化为至多$2log(n)$个区间操作，请使用常数尽可能小的数据结构（如标记永久化的线段树）来维护区间信息。

`dc`: dfs序时钟，剖分前需清零

`dd[x]`：`x`的深度

`df[x]`：`x`的父亲

`dh[x]`：`x`的重儿子

`ds[x]`：`x`的子树大小

`dt[x]`：`x`所在重链的头

`dl[x]`：`x`的dfs序，同时是以点为根的子树dfs序区间左端点

`dr[x]`：以`x`为根的子树dfs序区间右端点（闭）

`di[i]`：dfs序为i的点

```
vector<int> g[N];
int dd[N], df[N], dh[N], ds[N], dt[N], dl[N], dr[N], di[N], dc;
int dfs_hld1(int u, int f) {
    dd[u] = dd[f] + 1; df[u] = f;
    dh[u] = 0; ds[u] = 1;
    for (int v : g[u]) {
        if (v == f) continue;
        ds[u] += dfs_hld1(v, u);
        if (ds[v] > ds[dh[u]]) dh[u] = v;
    }
    return ds[u];
}

void dfs_hld2(int u, int t) {
    dl[u] = ++dc; di[dc] = u; dt[u] = t;
    if (dh[u]) dfs_hld2(dh[u], t);
    for (int v : g[u])
        if (v != df[u] && v != dh[u])
            dfs_hld2(v, v);
    dr[u] = dc;
}
```

```
//  Example:
dc = 0; dfs_hld1(r, 0); dfs_hld2(r, r);
```

求LCA

```
int lca(int u, int v) {
    for (; dt[u] != dt[v]; u = df[dt[u]])
        if (dd[dt[u]] < dd[dt[v]]) swap(u, v);
    return dd[u] < dd[v] ? u : v;
}
```

链操作

```
void some_operation(int u, int v, ...) {
    for (; dt[u] != dt[v]; u = df[dt[u]]) {
        if (dd[dt[u]] < dd[dt[v]]) swap(u, v);
        //  区间操作[dl[dt[u]], dl[u]]
    }
    if (dd[u] > dd[v]) swap(u, v);
    //  区间操作[dl[u], dl[v]], 若修改边权则将左端点+1
}
```

子树操作

```
void some_operation(int u, ...) {
    //  区间操作 [dl[u], dr[u]]
}
```

长链剖分

Splay

基本定义

```
int ch[3][N], sz[N], nc;
int *ls = ch[0], *rs = ch[1], *fa = ch[2];
int val[N];

int id(int x) { return ch[1][fa[x]] == x; }
bool isr(int x) { return !fa[x]; }

int gn(int v) {
    int p = ++nc;
    ls[p] = rs[p] = fa[p] = 0;
    sz[p] = 1; val[p] = v;
    //  ...
    return p;
}

void update(int x, ...) {
    //  ...
}

void push_up(int x) {
    sz[x] = sz[ls[x]] + sz[rs[x]] + 1;
    //  ...
}

void push_down(int x) {
    if (ls[x]) update(ls[x], ...);
    if (rs[x]) update(rs[x], ...);
    //  ...
}

void rot(int x) {
    int y = fa[x], z = fa[y], o = id(x), w = ch[!o][x];
    if (!isr(y)) ch[id(y)][z] = x; fa[x] = z;
    ch[!o][x] = y; fa[y] = x;
```

```
    ch[o][y] = w; if (w) fa[w] = y;
    push_up(y); push_up(x);
}

void splay(int x) {
    for (int y; !isr(x); rot(x))
        if(!isr(y=fa[x])) rot(id(x)^id(y)?x:y);
}
```

序列维护

注：所有不返回值的`find`在调用完后所求节点即为根。

求`rank`可在`find`后直接使用`sz[ls[r]]+1`。

建树

```
int build(const vector<int>& v, int lb, int rb) {
    if (lb == rb) return 0;
    int mid = (lb + rb) >> 1;
    int x = gn(v[mid]);
    ls[x] = build(v, lb, mid); if (ls[x]) fa[ls[x]] = x;
    rs[x] = build(v, mid + 1, rb); if (rs[x]) fa[rs[x]] = x;
    push_up(x);
    return x;
}
```

```
//  Example:
int n; cin >> n;
vector<int> vv(n);
for (int& v : vv) cin >> v;
int r = build(vv, 0, v.size());
```

查找序列第k个节点

```
void find_kth(int& r, int k) {
    int x = r;
    while (1) {
        push_down(x); int cnt = sz[ls[x]];
        if (cnt >= k) x = ls[x];
        else if (cnt == k - 1) break;
        else x = rs[x], k -= cnt + 1;
    }
    splay(x); r = x;
}
```

插入元素

在第$k-1$个元素后插入值为v的元素。

```
void insert_kth(int& r, int k, int v) {
    int* p = &r, x;
    while (*p) {
        push_down(x = *p); int cnt = sz[ls[x]];
        if (k <= cnt + 1) p = &ls[x];
        else p = &rs[x], k -= cnt + 1;
    }
    int y = *p = gn(v); fa[y] = x;
    splay(y); r = y;
}
```

删除根节点

删除策略：

1. 若 `x` 有左儿子，则将 `x` 的前驱旋至根，这会使得 `x` 没有左儿子。
2. 若 `x` 是根，则将 `x` 的右儿子置为根。
3. 否则 `x` 有父亲，则将 `x` 的右儿子设为 `x` 的父亲的右儿子。

```
void erase(int& r) {
    int x = r; push_down(x);
    if (ls[x]) {
        int y = ls[x]; push_down(y);
        while (rs[y]) push_down(y = rs[y]);
        splay(y); r = y;
    }
    if (isr(x)) {
        r = rs[x]; if (rs[x]) fa[rs[x]] = 0;
    }
    else {
        int w = rs[x], y = fa[x];
        ch[id(x)][y] = w; if (w) fa[w] = y;
        push_up(y);
    }
}
```

查找区间

返回代表区间 `[lb,rb]` 的点。

```
int find_range(int& r, int lb, int rb) {
    assert(lb <= rb);
    int n = sz[r], x = r;
    if (lb != 1 && rb != n) {
        find_kth(r, rb + 1); int rp = r;
        find_kth(r, lb - 1); int lp = r;
        if (fa[rp] != lp) rot(rp); x = ls[rs[r]];
    }
    else if (rb != n) find_kth(r, rb + 1), x = ls[r];
    else if (lb != 1) find_kth(r, lb - 1), x = rs[r];
    return x;
}
```

修改区间

将...中的tag打到区间$[lb,rb]$上

```
void modify_range(int& r, int lb, int rb, ...) {
    int x = find_range(r, lb, rb); update(x, ...);
    while (!isr(x)) push_up(x = fa[x]);
}
```

插入区间

在第$k-1$个元素后插入区间。

```
void insert_range(int& r, int x, int k) {
    if (!r) r = x;
    else {
        int n = sz[r];
        if (k == 1) find_kth(r, 1), ls[r] = x, push_up(fa[x] = r);
        else if(k == n + 1) find_kth(r, n), rs[r] = x, push_up(fa[x] = r);
        else {
            find_kth(r, k); find_kth(r, k - 1);
            ls[rs[r]] = x; push_up(fa[x] = rs[r]); push_up(r);
        }
    }
}
```

删除区间

删除区间$[lb,rb]$

```
void erase_range(int& r, int lb, int rb) {
    int x = find_range(r, lb, rb);
    if (!isr(x)) {
        ch[id(x)][fa[x]] = 0;
        //  recycle(x)
        while (!isr(x)) push_up(x = fa[x]);
    }
    else r = 0;
}
```

二叉搜索树扩展

查找第一个键值大于等于v的节点

找不到则返回`false`。

```
bool find_lwr(int& r, int v) {
    int x = r, y = -1;
    while (x) {
        push_down(x);
        if (val[x] >= v) y = x;
        x = ch[val[x] < v][x];
    }
    if (y != -1) splay(y), r = y;
    return y != -1;
}
```

注：差不多就是`multiset<int>::lower_bound`

查找第一个键值大于v的节点

```
bool find_lwr(int& r, int v) {
    int x = r, y = -1;
    while (x) {
        push_down(x);
        if (val[x] > v) y = x;
        x = ch[val[x] <= v][x];
    }
    if (y != -1) splay(y), r = y;
    return y != -1;
}
```

注：差不多就是`multiset<int>::upper_bound`

新建值为v的节点并插入

```
void insert(int& r, int y) {
    int x = 0, *p = &r;
    while (*p) {
        push_down(x = *p);
        p = &ch[val[x]<=val[y]][x];
    }
    fa[y] = x; *p = y; splay(y); r = y;
}
```

分裂与合并

`split`返回值域为[v,+inf)的splay，原树值域变为(-inf,v)。

`merge`将两棵值域至多交于一个点的splay合并，`r`中最大值小于等于`x`中最小值。

```
int split(int& r, int v) {
    if (!find_lwr(r, v)) return 0;
    int x = ls[r], y = r;
    fa[x] = ls[y] = 0;
    r = x; return y;
}

void merge(int& r, int x) {
    if (!r) { r = x; return; }
    int y = r; while (rs[y]) push_down(y), y = rs[y];
    splay(y); r = y;
    rs[r] = x; fa[x] = r;
}
```

区间翻转扩展

```
int rev[N];

int gn(int v) {
    rev[p] = 0;
    //  ...
}

inline void update(int x, int v) {
    if (v) {
        swap(ls[x], rs[x]);
        rev[x] ^= 1;
    }
}
```

注：`update`中其他与左右相关的（如左侧最大子段和）也需要swap

```
//  Example: Reverse Range[lb, rb]
modify_range(r, lb, rb, 1);
```

LCT

在带区间反转的Splay上修改：

1. 重定义`isr`
2. `splay`前将`x`到根路上所有标记下推

```
inline bool isr(int x) { return ch[id(x)][fa[x]] != x; }

void splay(int x) {
    static int s[N], t; int z = s[++t] = x;
    while (!isr(z)) s[++t] = z = fa[z];
    while (t) push_down(s[t--]);
    //...
}
```

access

建立`x`到根的链

```
void access(int x) {
    for (int y = 0; x; x = fa[y = x])
        splay(x), rs[x] = y, push_up(x);
}
```

find_root

找到`x`所在LCT的根

```
int find_root(int x) {
    access(x); splay(x); push_down(x);
    while (ls[x]) push_down(x = ls[x]);
    splay(x); return x;
}
```

make_root

将`x`设为所在LCT的根

```
void make_root(int x) {
    access(x); splay(x);
    update(x, 1);
}
```

split

建立`x`到`y`的链，调用后`y`为那条链对应splay树的根

```
void split(int x, int y) {
    make_root(x); access(y); splay(y);
}
```

link

建边`(x,y)`，若之前`x`与`y`在同一LCT中则返回`false`，否则建边并返回`true`。

```
bool link(int x, int y) {
    make_root(x);
    if (find_root(y) == x) return false;
    fa[x] = y; return true;
}
```

cut

删边`(x,y)`，若边`(x,y)`不存在则返回`false`，否则删边并返回`true`。

```
bool cut(int x, int y) {
    make_root(x);
    if (find_root(y) != x || fa[y] != x || ls[y]) return false;
    fa[y] = rs[x] = 0; push_up(x); return true;
}
```