Link Cut Tree

在静态的树结构中，树剖是维护路径/子树修改的实用方法，但树剖无法处理动态的树结构。而LCT正是维护动态树结构的强大工具。LCT擅长维护树链信息，能很容易地实现路径修改+路径查询。

LCT也能处理一些子树相关的问题，稍加修改即可支持单点修改+子树查询，前提是维护的信息可减。但更加通用的子树修改+子树查询，基本只能用LCT的升级版TopTree来解决。

LCT的所有操作都是 $\mathcal{O}(\log n)$ 的，但是由于自带的常数巨大无比，因此可以把LCT的复杂度近似看作 $\mathcal{O}(\log^2 n)$。

板子

> **三年竞赛一场空，不判自环见祖宗**

```
template<class Info,class Tag,int MAX_SIZE,
         bool CHECK_LINK = 0,bool CHECK_CUT = 0,bool ASSERT = 0>
struct LinkCutTree {
    #define lch tr[u].ch[0]
    #define rch tr[u].ch[1]
    #define wch(u) (tr[tr[u].p].ch[1]==u)

    struct Node {
        int ch[2],p;
        bool rev;
        Info info;
        Tag tag;
```

```
        void update(const Tag &x) {
            info.update(x);
            tag.update(x);
        }
    };
    array<Node, MAX_SIZE> tr;
    array<int, MAX_SIZE> stk;

    bool is_root(int u) {
        return tr[tr[u].p].ch[0]!=u&&tr[tr[u].p].ch[1]!=u;
    }

    void pushup(int u) {
        tr[u].info.pushup(tr[lch].info,tr[rch].info);
    }

    void pushrev(int u) {
        tr[u].info.reverse();
        swap(lch,rch);
        tr[u].rev^=1;
    }

    void pushdn(int u) {
        if(tr[u].rev) {
            if(lch) pushrev(lch);
            if(rch) pushrev(rch);
            tr[u].rev=0;
        }
        if(lch) tr[lch].update(tr[u].tag);
        if(rch) tr[rch].update(tr[u].tag);
        tr[u].tag.clear();
    }

    void rotate(int x) {
        int y=tr[x].p,z=tr[y].p,k=wch(x);
        if(!is_root(y)) tr[z].ch[wch(y)]=x;
        tr[y].ch[k]=tr[x].ch[!k],tr[tr[y].ch[k]].p=y;
        tr[x].ch[!k]=y,tr[y].p=x,tr[x].p=z;
        pushup(y),pushup(x);
    }

    void splay(int u) {
        int top=0,fa=u;
        stk[++top]=fa;
        while(!is_root(fa)) stk[++top]=fa=tr[fa].p;
        while(top) pushdn(stk[top--]);
        for(;!is_root(u);rotate(u))
            if(!is_root(fa=tr[u].p)) rotate(wch(u)==wch(fa)?fa:u);
    }

    int access(int u) {
        int v=0;
        for(;u;v=u,u=tr[u].p)
            splay(u),rch=v,pushup(u);
```

```
        return v;
    }

    void make_root(int u) {
        access(u);
        splay(u);
        pushrev(u);
    }

    int split(int u,int v) {
        make_root(u);
        access(v);
        splay(v);
        return v;
    }

    int find_root(int u) {
        access(u);
        splay(u);
        while(lch) pushdn(u),u=lch;
        splay(u);
        return u;
    }

    bool same(int u,int v) {
        make_root(u);
        return find_root(v)==u;
    }

    bool link(int u,int v) {
        make_root(u);
        if(CHECK_LINK&&find_root(v)==u)
            return assert(!ASSERT),0;
        tr[u].p=v;
        return 1;
    }

    bool cut(int u,int v) {
        make_root(u);
        if(CHECK_CUT&&!(find_root(v)==u&&rch==v&&!tr[v].ch[0]))
            return assert(!ASSERT),0;
        else access(v),splay(u);
        rch=tr[v].p=0;
        pushup(u);
        return 1;
    }

    int lca(int u,int v) {
        access(u);
        return access(v);
    }

    int lca(int rt,int u,int v) {
        make_root(rt);
```

```
        return lca(u,v);
    }

    void modify(int u,const Tag &x) {
        if(!is_root(u)) splay(u);
        tr[u].update(x);
    }

    Info &info(int u) {
        return tr[u].info;
    }

    #undef lch
    #undef rch
    #undef wch
};

struct Tag {

    void update(const Tag &x) {

    }

    void clear() {

    }
};

struct Info {

    //* lch+parent+rch
    void pushup(const Info &l,const Info &r) {

    }

    void update(const Tag &x) {

    }

    void reverse() {}
};

LinkCutTree<Info,Tag,N> lct;
```

调试&卡常

关于常数问题，由于 `Info` 中的 `pushup` 函数通常是LCT中调用最为频繁的函数，因此这部分的细节处理就很关键。具体来说：

- 在维护MST时，手写 `if else` 通常会比开一个 `array/vector` 然后 `sort` 快一倍。然后将边信息改为保存边id也能些微减小常数，但是用起来不如直接暴力存边直观。
- 有时并不需要用的Tag或者存在一些空函数，把这些东西删掉能减少一部分常数。

路径修改+路径查询

这是LCT最基础的用法，没有太多可讲的点。需要注意修改的时候，点必须 `splay` 到根。

```
void modify(int u,const Tag &x) {
    if(!is_root(u)) splay(u);
    tr[u].update(x);
}
```

然后如果维护的信息具有方向性，记得合理实现 `reverse` 函数。

单点修改+子树查询

在 `Info` 中将要维护的信息分离为虚/实两部分，然后添加两个新方法，用来处理添加/删除虚信息。要求信息必须可减。

单点加法+子树求和的例子。常见的用法还有维护子树大小，也是类似的写法。

```
struct Info {
    LL val=0;
    LL sum=0,vsum=0;

    void pushup(const Info &l,const Info &r) {
        // 左右子树的和+所有虚子树的和+自己的value
        sum=l.sum+r.sum+vsum+val;
    }

    // 添加一个新的虚子树
    void add(const Info &x) {
        vsum+=x.sum;
    }

    // 删去一个虚子树
    void sub(const Info &x) {
        vsum-=x.sum;
    }
};
```

然后修改以下几个函数。

`access` 的过程中产生了边的虚实变化，因此需要修改。

```
int access(int u) {
    int v=0;
    for(;u;v=u,u=tr[u].p) {
        splay(u);
        // 实 -> 虚，添加虚子树
        if(rch) tr[u].info.add(info(rch));
```

```
            // 虚 -> 实，删去虚子树
            if(v) tr[u].info.sub(info(v));
            rch=v,pushup(u);
        }
        return v;
    }
```

`link` 添加了一个新的虚儿子 `u` 到 `p`，因此也需要修改。注意必须要先 `make_root` 再做修改。

```
bool link(int p,int u) {
    make_root(u);
    if(CHECK_LINK&&find_root(p)==u)
        return assert(!ASSERT),0;
    make_root(p);
    tr[p].info.add(info(u));
    tr[u].p=p;
    pushup(p);
    return 1;
}
```

而在 `cut` 中，我们断开的是实边，因此不需要做修改，`pushup` 会维护信息的变化。

务必注意，和常规的LCT不同，在修改之前，必须先 `access` 然后再 `splay`，以保证点为整颗树的根。统计子树信息时，也必须保证为树根。

```
// modify
lct.access(u);
lct.splay(u);
lct.info(u)={...};

// query
lct.access(u);
lct.splay(u);
cout<<lct.info(u).sum<<endl;
```

如果要查询指定子树的信息，而不是整棵树的信息，例如 u 点以 p 作为为父节点时 u 的子树和。那么我们先 `cut` 掉 (u,p)，查询完之后再 `link` 回去即可。

```
lct.cut(u,p);
// 这里不需要再转到根了，因为cut函数保证了cut完之后u,p都是根
cout<<lct.info(u).sum<<endl;
lct.link(u,p);
```

维护MST

维护MST几乎可以说是LCT最常见的应用。

由于LCT不方便直接操作边，我们可以使用虚点的技巧来将边信息保存为点信息。即将 (x,y) 边拆分为 $(x,z),(z,y)$，其中 z 为新建的虚点，将边权保存为 z 的点权。x,y 为非边点，点权为 ∞ 或 0。

```
for(int i=1;i<=m;i++) {
    int u,v,val;
    cin>>u>>v>>val;
    int w=i+n;
    lct.info(w)={val,val,{u,v,w},{u,v,w}};
    add_edge(u,v,w,val);
}
```

最小生成树

以最小生成树为例，我们在LCT中维护以下信息：

- 节点边权
- 子树最大边权
- 节点所代表的边
- 子树最大边权所代表的边

```
struct MaxInfo {
    int v=0,maxv=0;
    Edge e,maxe;

    void pushup(const MaxInfo &l,const MaxInfo &r) {
        if(l.maxv>=r.maxv) maxv=l.maxv,maxe=l.maxe;
        else maxv=r.maxv,maxe=r.maxe;
        if(v>maxv) maxv=v,maxe=e;
    }
};
```

`Edge` 通常为 `tuple<int,int,int>` 或者 `tuple<int,int,int,int>`，取决于是否要保存边权。

在维护最小生成树的过程，需要根据是否已经连通来分类讨论。

```
// 添加(u,v)边，虚点为w，边权为val
auto add_edge=[&](int u,int v,int w,int val) {
    // 如果已经连通，那么需要判断环上的最大边权是否比val大
    if(lct.same(u, v)) {
        int rt=lct.split(u, v);
        if(lct.info(rt).maxv>val) {
            auto [x,y,z]=lct.info(rt).maxe;
            lct.cut(x, z);
            lct.cut(y, z);
            lct.link(u, w);
            lct.link(v, w);
        }
    }
```

```
    else {
        lct.link(u, w);
        lct.link(v, w);
        cnt++;
    }
};
```

删边则比较简单。

```
// 删除边(u,v)，判断其中一个点是否与虚点连通即可
auto del_edge=[&](int u,int v,int w) {
    if(lct.same(u, w)) {
        lct.cut(u, w);
        lct.cut(v, w);
        cnt--;
    }
};
```

最大生成树

将最小生成树对称过来即可。

```
struct MinInfo {
    int v=INF,minv=INF;
    Edge e,mine;

    void pushup(const MinInfo &l,const MinInfo &r) {
        if(l.minv<r.minv) minv=l.minv,mine=l.mine;
        else minv=r.minv,mine=r.mine;
        if(v<minv) minv=v,mine=e;
    }
};
```

处理加边。

```
auto add_edge=[&](int u,int v,int w,int val) {
    if(lct.same(u, v)) {
        int rt=lct.split(u, v);
        if(lct.info(rt).minv<val) {
            auto [x,y,z]=lct.info(rt).mine;
            lct.cut(x, z);
            lct.cut(y, z);
            lct.link(u, w);
            lct.link(v, w);
        }
    }
    else {
```

```
            lct.link(u, w);
            lct.link(v, w);
            cnt++;
        }
    };
```

复杂度 $\mathcal{O}(m\log m)$。

可撤销地维护MST

需要可撤销地维护MST，通常是问题的边有时效性，或者有加删边，需要使用线段树分治。

和DSU不同，LCT本身支持删除，所以实现上不需要特殊的技巧。

```
// 把边丢到线段树上
void add(int u,int x,int y,int l,int r,Edge val) {
    if(x>r||y<l) return;
    if(x<=l&&y>=r) seg[u].emplace_back(val);
    else {
        int mid=(l+r)/2;
        add(lch,x,y,l,mid,val);
        add(rch,x,y,mid+1,r,val);
    }
}

// 维护MST边权和的例子
LL sum;
void dfs(int u,int l,int r) {
    LL bak=sum;
    vector<Edge> del,add;

    for(auto [x,y,z,w]:seg[u]) {
        // 修改MST，把加删的边丢进del和add保存
    }

    if(l==r) ans[l]=sum;
    else {
        int mid=(l+r)/2;
        dfs(lch, l, mid);
        dfs(rch, mid+1, r);
    }

    // 这里务必按照逆序做，否则会导致加删不存在的边而导致RE
    while(add.size()) {
        auto [x,y,z,w]=add.back();
        lct.cut(x, z);
        lct.cut(y, z);
        tie(x,y,z,w)=del.back();
        lct.link(x, z);
        lct.link(y, z);
        add.pop_back();
```

```
            del.pop_back();
        }
        sum=bak;
    }
```

复杂度 $\mathcal{O}(m\log m\log t)$ 。t 为时间跨度。

维护树直径/重心

todo

维护割点割边

不会QAQ

LCT 问题集

> luogu LCT 题单

[SDOI2008]洞穴勘测

题意

给定一个有 n 个点的图，刚开始没有边。m 次操作。

- 加上一条边 (u,v)
- 删去一条边 (u,v)
- 询问 (u,v) 是否连通

使用lct维护动态连通性的模板题。

复杂度 $\mathcal{O}(m\log n)$。

实现

> 评测记录

```
bool same(int u,int v) {
    make_root(u);
    return find_root(v)==u;
}
```

也可以写成 `find_root(u)==find_root(v);`。

[NOI2014] 魔法森林

题意

给定一个 n 个点 m 条边的无向图。每条边有两个边权 a_i,b_i。定义两个阈值 A,B，当 $A \ge a_i$ 且 $B \ge b_i$ 时 i 号边存在，求最小的 $A+B$，使得 1 号点和 n 号点联通。

思路

lct维护加边最小生成树。

将所有边按照 a,b 的双关键字排序后按顺序添加，用LCT维护以 b 为边权的最小生成树即可。

关于正确性，这个过程中我们实际上枚举了 A 的上界，那么由于 A 的代价是确定的，我们只要使 B 尽可能小即可，而这样的 B 必定可以在MST中取到。证明起来和 kruskal 差不多，根据MST的性质，如果我们试图加入一条不在MST中的边，那么这条边的边权必然会不小于环上的最大边权，因此无论加上哪条边，都只可能使得 B 变得更大。

复杂度 $\mathcal{O}(m\log m)$。

实现

> 评测记录

```
void solve() {
    int n,m;
    cin>>n>>m;
    vector<tuple<int,int,int,int,int>> edg(m);
    for(auto &[a,b,u,v,w]:edg) cin>>u>>v>>a>>b;
    sort(edg.begin(),edg.end());

    for(int i=1;i<=m;i++) {
        auto &[a,b,u,v,w]=edg[i-1];
        w=i+n;
        lct.info(w)={b,i-1,b,i-1};
    }

    constexpr int INF=1e9;
    int ans=INF;

    for(auto [a,b,u,v,w]:edg) {
        if(lct.same(u,v)) {
            int rt=lct.split(u, v);
            int id=lct.info(rt).maxid;
            if(lct.info(rt).maxv>b) {
                auto [_,__,x,y,z]=edg[id];
                lct.cut(x, z);
                lct.cut(y, z);
                lct.link(u, w);
                lct.link(v, w);
            }
        }
        else {
            lct.link(u, w);
            lct.link(v, w);
        }
```

```
            if(lct.same(1, n)) ans=min(ans,lct.info(lct.split(1, n)).maxv+a);
        }

        cout<<(ans==INF?-1:ans)<<endl;
    }
```

[国家集训队] Tree II

树上路径修改+路径查询

题意

一棵 n 个点的树，每个点的初始权值为 1。
对于这棵树有 q 个操作，每个操作为以下四种操作之一：

- `+ u v c`：将 u 到 v 的路径上的点的权值都加上自然数 c；
- `- u1 v1 u2 v2`：将树中原有的边 (u_1, v_1) 删除，加入一条新边 (u_2, v_2)，保证操作完之后仍然是一棵树；
- `* u v c`：将 u 到 v 的路径上的点的权值都乘上自然数 c；
- `/ u v`：询问 u 到 v 的路径上的点的权值和，将答案对 51061 取模。

思路

LCT维护路径修改+路径查询板子题。

实现

> 评测记录

```
struct Tag {
    Mint mul=1,add=0;

    void update(const Tag &x) {
        if(x.mul.v!=1) {
            mul*=x.mul;
            add*=x.mul;
        }
        if(x.add.v) {
            add+=x.add;
        }
    }

    void clear() {
        mul=1;
        add=0;
    }
};

struct Info {
    int sz=0;
```

```
    Mint sum=0,val=0;

    // lch+parent+rch
    void pushup(const Info &l,const Info &r) {
        sz=l.sz+r.sz+1;
        sum=l.sum+r.sum+val;
    }

    // update info by lazy tag
    void update(const Tag &x) {
        if(x.mul.v!=1) {
            sum*=x.mul;
            val*=x.mul;
        }
        if(x.add.v) {
            sum+=x.add*sz;
            val+=x.add;
        }
    }
};
```

[Cnoi2019] 须臾幻境

题意

区间数连通块，强制在线。

你有一个无向图 $G(V,E)$, E 中每一个元素用一个二元组 (u,v) 表示。

现在把 E 中的元素排成一个长度为 $|E|$ 序列 A。

然后给你 q 个询问二元组 (l,r),

表示询问 图 $G'\left(V, \bigcup\limits_{i\in[l,r]} A_i\right)$ 的联通块的个数。

思路

我们先对问题做一些转化，对于一个询问区间，假设我们以及维护出了一个生成森林，边数为 m，那么连通块数量即为 $n-m$。

显然对于每个询问我们不能现算MST，否则会直接T飞，但是考虑到区间是静态的，即没有修改，我们可以考虑先进行预处理，然后用主席树之类的数据结构保存答案。

受区间数颜色 $\mathcal{O}(n\log n)$ 做法的启发，我们可以考虑给每条边赋予一个时间戳 t_i，然后从左向右遍历边数组加边，维护出一个以时间戳为边权的最大生成森林。

考虑正确性，对于区间右端点为 r 的一个最大生成森林，我们必然是尽可能使边的时间尽可能大，即尽可能靠近 r。这样对于一个 $[l,r]$ 的询问，答案就等于森林中时间戳在 $[l,r]$ 区间中的边的数量，因为我们最大限度地使用了 $[l,r]$ 中的边。

于是我们可以使用一颗可持久化权值线段树保存每个 r 端点的边权集合即可。

复杂度 $\mathcal{O}((n+q)\log n)$。

实现

> 评测记录

注意大坑自环!

```
// 维护MST,更新线段树
for(int i=1;i<=m;i++) {
    auto [u,v,w]=edg[i];
    if(u==v) {
        rootid[i]=rootid[i-1];
        continue;
    }
    if(lct.same(u, v)) {
        int t=lct.info(lct.split(u, v)).mint;
        auto [x,y,_]=edg[t];
        lct.cut(x, t);
        lct.cut(y, t);
        sgt.update(t, SgtInfo{0});
    }
    lct.link(u, w);
    lct.link(v, w);
    rootid[i]=sgt.update(w, SgtInfo{1});
}

// 答案就是n-边数
while(q--) {
    int l,r;
    cin>>l>>r;
    get_query(l, r);
    int res=sgt.query(rootid[r],l,r).sum;
    ans=n-res;
    cout<<ans<<endl;
}
```

[BJOI2014] 大融合

题意

给定一颗 n 个点的树。q 次操作。

- 添加一条边 (x,y)
- 询问有多少路径经过了边 (x,y)

思路

答案就是 x 子树大小乘上 y 子树大小。使用上面提到的LCT虚子树的技巧维护子树大小即可。

查询的时候记得 `cut` 掉 (x,y) 然后再 `link` 回去。

复杂度 $\mathcal{O}(q\log n)$。

实现

> 评测记录

[TJOI2015] 旅游

题意

给定一棵带点权的树，q次询问，每次询问树上从u到v的一条路径，要求选路径上两个点x,y（需要保证方向和u->v一致），使得y-x的权值最大，求最大值。之后将这条路径上的点权全部加上x。

思路1 大力树剖！

由于这道题给定的树是静态的，所以其实更好想到的做法是树剖+线段树。

线段树维护如下几个值：

```
int minn=INF,maxx=0,ltor=0,rtol=0;
```

分别是最小值，最大值，从左到右的答案最大值，从右到左的答案最小值。由于题目有方向的限制，所以答案也要维护两个方向。

信息合并也很自然：

```
friend Info operator+(const Info &l,const Info &r) {
    Info res;
    res.minn=min(l.minn,r.minn);
    res.maxx=max(l.maxx,r.maxx);
    res.ltor=max({l.ltor,r.ltor,r.maxx-l.minn});
    res.rtol=max({l.rtol,r.rtol,l.maxx-r.minn});
    return res;
}
```

由于树剖出来的几个段在线段树中不一定连续，所以还要处理跨段间的信息合并。

假设 $lca(u,v)=p$，稍微分类讨论一下：

- 如果 $p=u$ 或者 $p=v$，这种情况就是深度严格单调的一条路径，可以很轻松地处理。
- 否则可以拆分成 $u\to p,p\to v$ 两段，按照上述流程做两遍然后再合并两段间的信息即可。

复杂度 $\mathcal{O}(n\log^2 n)$。

实现

> 评测记录

```
int u,v,w;
cin>>u>>v>>w;

int p=hpd::lca(u,v);
int ans=0;

// 处理路径深度单调的情况
auto work=[&](int u,int v) {
    auto &&seg=hpd::decompose(u,v);
    sort(seg.begin(),seg.end());
    int minn=INF,maxx=0;
    bool rev=seg.front().first!=hpd::id[u];
    if(rev) reverse(seg.begin(),seg.end());

    auto get=[&](auto node) {
        if(!rev) return node.ltor;
        return node.rtol;
    };

    for(auto [l,r]:seg) {
        auto i=sgt.query(l,r);
        ans=max(ans,get(i));
        ans=max(ans,i.maxx-minn);
        minn=min(minn,i.minn);
        maxx=max(maxx,i.maxx);
    }
    return pair{minn,maxx};
};

// 分类讨论
if(u==p||v==p) work(u,v);
else {
    auto [minn,_]=work(u,p);
    auto [__,maxx]=work(p,v);
    ans=max(ans,maxx-minn);
}

for(auto [l,r]:hpd::decompose(u,v)) sgt.modify(l,r,Tag{w});
cout<<ans<<endl;
```

思路2 直接上LCT!

lct中维护信息：

```
int val=0,minn=INF,maxx=0,lmax=0,rmax=0;
```

分别为点权，最小值，最大值，左max-右min的最大值（对应上面的rtol），右max-左min的最大值（对应ltor）。

信息合并：

```
void pushup(const Info &l,const Info &r) {
    minn=min({l.minn,r.minn,val});
    maxx=max({l.maxx,r.maxx,val});
    lmax=max({l.lmax,r.lmax,max(l.maxx,val)-min(r.minn,val)});
    rmax=max({l.rmax,r.rmax,max(r.maxx,val)-min(l.minn,val)});
}
```

由于存在方向的限制，所以在区间翻转时也要翻转对应的节点信息。

> 然后这里务必要保证pushup的时候，子节点已经翻转完成。

```
// in lct
void pushrev(int u) {
    tr[u].info.reverse();
    swap(lch,rch);
    tr[u].rev^=1;
}

// in info
void reverse() {
    swap(lmax,rmax);
}
```

处理查询非常简单：

```
int u,v,w;
cin>>u>>v>>w;
int rt=lct.split(v, u);
cout<<lct.info(rt).lmax<<endl;
lct.modify(rt, Tag{w});
```

整体代码比树剖短不少，复杂度 $\mathcal{O}(n\log n)$，但是实际跑起来比树剖慢。

> 评测记录

[USACO18FEB] New Barns P

题意

给你一棵树，初始没有节点。你需要支持两种操作：

- `B p` 表示新建一个节点，将它与 p 节点连接；若 $p=-1$，则表示不与其它节点相连
- `Q k` 表示查询在 k 节点所在的连通块中，距它最远的点的距离。这里距离的定义是两点间经过的边数。

思路

问题可以转化为用lct维护树直径。当然也有离线的做法：把边离线下来然后用倍增维护lca和路径长度。

处理查询

假设当前连通块的直径两端点已知为 x,y，查询的点为 u，那么可以分类讨论：

- 如果 u 在 x,y 的路径上，根据直径的性质，答案就等于 $\max len(u,x),len(u,y)-1$ 。其中 len 是计算路径点数而不是边数，所以需要 -1。
- 否则，记 u 与 x,y 路径交于 z，同样根据直径的性质，答案等于 $\max len(u,x),len(u,y)-1+len(u,z)-1$ 。

处理修改

现在考虑如何维护直径，假设新建的点标号为 idx，连到 u 上。

如果 $u=-1$，那么简单地新建点即可。

否则，同样假设已知 x,y,z，分类讨论：

- 如果 $u=x$，或者 $u=y$，也就是直径末端添加一个点，那么之间将直径末端的点更新为 idx 即可。
- 否则，考虑什么时候会更新直径，因为 $len(z,x),len(z,y)$ 都是极长的，所以只有当 $len(z,x)=len(z,u)$ 或者 $len(z,y)=len(z,u)$ 时才会更新。

此外，这个问题可以拓展到合并两颗树，可以推出一个类似的结论，即新树的直径两端点必然是从原本的四个点中选出来的。本题可以看作第二颗树大小为1的特化版本。

时间复杂度 $\mathcal{O}(n\log n)$。

实现

评测记录

```
void solve() {
    int idx=0;
    vector<int> id(1);
    vector<pair<int,int>> ed(1);

    int q;
    cin>>q;
    while(q--) {
        char op;
        int u;
        cin>>op>>u;

        auto len=[&](int u,int v) {
            return lct.info(lct.split(u,v)).sz;
        };

        if(op=='B') {
            idx++;
            if(u==-1) {
```

```
                    id.push_back(idx);
                    ed.emplace_back(idx,idx);
                }
                else {
                    id.push_back(id[u]);
                    auto &[x,y]=ed[id[u]];
                    if(u==x) x=idx;
                    else if(u==y) y=idx;
                    else {
                        int z=lct.lca(x,y,u);
                        if(len(z,x)==len(z,u)) x=idx;
                        else if(len(z,y)==len(z,u)) y=idx;
                    }
                    lct.link(u, idx);
                }
            }
            else {
                auto [x,y]=ed[id[u]];
                int z=lct.lca(x,y,u);
                int ans=max(len(x,z),len(y,z));
                if(u!=z) ans+=lct.info(lct.split(u,z)).sz-1;
                cout<<ans-1<<endl;
            }
        }
    }
```

[WC2006] 水管局长

题意

给定 n 个点 m 条边的简单无向图，q 次操作，分为两种类型：

- 删除边 (u,v)
- 查询 $u \to v$ 的一条路径，使得路径权值最大值最小。求最小值。

$n=10^3, m=q=10^5$ 。

思路

先考虑没有修改操作怎么做。

我们依次加入每条边，假设当前图是一个森林，如果加入一条新边时形成了一个环，那么将环上权最大的边删去必定不会使得任意一条路径的答案变差。因为如果一条路径原先经过了最大边，现在只要走环的另一侧必定不会使得答案变差。

观察这个构造过程，实际上最终得到的图就最小生成树/森林。因此我们只需要用LCT维护MST即可。

现在再考虑修改，由于只有删除而没有添加，所以只需要将询问离线按照时间逆向处理，删就变成了增，问题就变得非常裸了。

时间复杂度 $\mathcal{O}(m \log m)$。

实现

> 评测记录

```
struct Info {
    int t,id;
    int maxt,maxid;

    //* lch+parent+rch
    void pushup(const Info &l,const Info &r) {
        vector v({pair(l.maxt,l.maxid),pair(r.maxt,r.maxid),pair(t,id)});
        sort(v.begin(),v.end());
        tie(maxt,maxid)=v.back();
    }
};

LinkCutTree<Info,Tag,N+M> lct;

void solve() {
    int n,m,q;
    cin>>n>>m>>q;
    vector<tuple<int,int,int>> edg(1);

    auto add_edge=[&](int u,int v,int t) {
        int w=edg.size();
        lct.info(w)={t,w,t,w};
        edg.emplace_back(u,v,w);
        if(lct.same(u, v)) {
            int rt=lct.split(u, v);
            int id=lct.info(rt).maxid;
            int tt=lct.info(rt).maxt;
            auto [x,y,z]=edg[id];
            if(tt>t) {
                lct.cut(x, z);
                lct.cut(y, z);
                lct.link(u, w);
                lct.link(v, w);
            }
        }
        else {
            lct.link(u, w);
            lct.link(v, w);
        }
    };

    map<pair<int,int>,int> mp;
    for(int i=1;i<=m;i++) {
        int u,v,t;
        cin>>u>>v>>t;
        if(u>v) swap(u,v);
        u+=M,v+=M;
        mp[{u,v}]=t;
```

```
    }

    vector<int> ans;
    vector<tuple<int,int,int,int>> qry(q);
    for(auto &[k,u,v,t]:qry) {
        cin>>k>>u>>v;
        if(u>v) swap(u,v);
        u+=M,v+=M;
        if(k==2) {
            t=mp[{u,v}];
            mp.erase({u,v});
        }
    }
    for(auto [_,t]:mp) {
        auto [u,v]=_;
        add_edge(u, v, t);
    }

    reverse(qry.begin(),qry.end());
    for(auto [k,u,v,t]:qry) {
        if(k==1) ans.emplace_back(lct.info(lct.split(u,v)).maxt);
        else add_edge(u, v, t);
    }

    reverse(ans.begin(),ans.end());
    for(int x:ans) cout<<x<<endl;
}
```

[ZJOI2012] 网络

思路

水题，开10个LCT模拟即可。

实现

> 评测记录

最小差值生成树

题意

给定一个点标号从 1 到 n 的、有 m 条边的无向图，求边权最大值与最小值的差值最小的生成树。图可能存在自环。

思路

将边按照边权排序，枚举最小权值然后双指针，用LCT维护最大生成森林。每当边数达到 $n-1$ 时更新答案。

虽然有删边操作，但是由于这个过程是单调/贪心的，即被删掉的边不可能再加回来，所以并不需要线段树分治。

复杂度 $\mathcal{O}(m\log m)$。

实现

> 评测记录

注意处理自环。

```
struct Info {
    int val=INF,minv=INF;
    tuple<int,int,int> edg,mine;

    void pushup(const Info &l,const Info &r) {
        vector<pair<int,tuple<int,int,int>>>
        v({{l.minv,l.mine},{r.minv,r.mine},{val,edg}});
        sort(v.begin(),v.end());
        tie(minv,mine)=v.front();
    }
};

LinkCutTree<Info,Tag,N+M> lct;
vector<tuple<int,int,int>> edg[W];

void solve() {
    int n,m,maxw=0;
    cin>>n>>m;
    for(int i=1;i<=m;i++) {
        int u,v,w;
        cin>>u>>v>>w;
        if(u==v) continue;
        maxw=max(maxw,w);
        u+=M,v+=M;
        edg[w].emplace_back(u,v,i);
        lct.info(i)={w,w,edg[w].back()};
    }

    int ans=INF;
    for(int l=0,r=0,cnt=0;l<maxw;l++) {
        auto add_edge=[&](int u,int v,int w,int val) {
            if(lct.same(u, v)) {
                int rt=lct.split(u, v);
                if(lct.info(rt).minv<val) {
                    auto [x,y,z]=lct.info(rt).mine;
                    lct.cut(x, z);
                    lct.cut(y, z);
                    lct.link(u, w);
                    lct.link(v, w);
                }
            }
            else {
                lct.link(u, w);
                lct.link(v, w);
                cnt++;
```

```
            }
        };

        auto del_edge=[&](int u,int v,int w) {
            if(lct.same(u, w)) {
                lct.cut(u, w);
                lct.cut(v, w);
                cnt--;
            }
        };

        for(auto [x,y,z]:edg[l]) del_edge(x, y, z);
        while(cnt<n-1&&r<maxw) {
            r++;
            for(auto [x,y,z]:edg[r]) add_edge(x, y, z, r);
        }

        if(cnt<n-1) break;
        else ans=min(ans,r-l-1);
    }

    cout<<ans<<endl;
}
```

连环病原体

题意

给定 m 条边，当由一个区间中的边构成的导出子图存在环时，则称区间是一个加强区间。

求每条边在多少个加强区间中出现过。

思路

几乎和上一道题的思路一模一样，枚举区间左端点，用双指针找到最近的一个会形成环右端点，然后就可以得到以 l 作为左端点的最短加强区间 $[l,r]$。并且 $[l,r] \to [l,m]$ 都是加强区间。

设 $len = m - r + 1$，计算加强区间对每条边的贡献：

- $[l,r]$ 中的边，答案加上 len
- $r+1$ 号边，答案加上 $len-1$
- $\cdots$
- m 号边，答案加上 1

可以用二重差分/前缀和来处理贡献计算。

时间复杂度 $\mathcal{O}(m \log m)$。

实现

> 评测记录

因为这道题只需要处理连通性，所以LCT中不需要维护额外的信息。

```
void solve() {
    int m;
    cin>>m;
    vector<tuple<int,int,int>> edg;
    vector<LL> d1(m+2),d2(m+2),ans(m+2);
    for(int i=1;i<=m;i++) {
        int u,v;
        cin>>u>>v;
        u+=M,v+=M;
        edg.emplace_back(u,v,i);
    }

    for(int l=0,r=0,cnt=0;l<m;l++) {
        bool loop=0;
        while(!loop&&r<m) {
            auto [u,v,w]=edg[r];
            if(lct.same(u, v)) loop=1;
            else {
                lct.link(u, w);
                lct.link(v, w);
                r++;
            }
        }

        if(!loop) break;
        auto [u,v,w]=edg[l];
        lct.cut(u, w);
        lct.cut(v, w);

        int len=m-r;
        d1[m-1+2]++;
        d1[r-1+2]--;
        d1[l-1+2]-=len;
        d1[l-2+2]+=len;
    }

    reverse(d1.begin(),d1.end());
    partial_sum(d1.begin(),d1.end(),d2.begin());
    partial_sum(d2.begin(),d2.end(),ans.begin());
    reverse(ans.begin(),ans.end());

    for(int i=2;i<=m+1;i++) cout<<ans[i]<<' ';
}
```

变化的道路

题意

给定 n 个点带边权的一颗树，附加 m 条可变的边，第 i 条可变边经在 $[l_i, r_i]$ 时间段存在。

询问每个时间段的MST边权和。

$n \le 5 \times 10^4, m \le 3 \times 32766$ 。时间范围固定为 32766。

思路

前置知识：线段树分治。

和上面 最小差值生成树 那道题不同，这题一条边被删除之后，我们可能又会将其加入，即LCT需要支持撤销操作。而LCT显然是不能随便撤销的，否则可能会引入当前时间段不存在的边。因此我们需要使用线段树分治来保证LCT撤销操作在时间上的正确姓。

套完线段树分治之后，这题就很裸了。对于撤销操作，只需要记录一下每条边加入时删除的边保存到栈中，回溯时倒着做即可。

注意，对于撤销操作，必须按照栈顺序做，如果直接存`vector`然后`for`过去，会导致LCT操作不存在的边。

```
3
1 2 50
1 3 20
2
2 3 30 1 3
2 1 10 1 3
```

设时间为 d，复杂度 $\mathcal{O}((n+m)\log(n+m)\log d)$ 。

实现

> 评测记录

```
struct Info {
    int val,maxv;
    int edg,maxe;

    // 这里pushup最好不要用vector直接sort，TLE了半天qwq
    void pushup(const Info &l,const Info &r) {
        if(l.maxv>=r.maxv) maxv=l.maxv,maxe=l.maxe;
        else maxv=r.maxv,maxe=r.maxe;
        if(val>maxv) maxv=val,maxe=edg;
    }
};

LinkCutTree<Info,N+N+M> lct;

#define lch (u<<1)
#define rch (u<<1|1)
vector<Edge> seg[D*4+10],edge(1);
LL ans[D+10];

void add(int u,int x,int y,int l,int r,Edge val) {
```

```
    if(x>r||y<l) return;
    if(x<=l&&y>=r) seg[u].emplace_back(val);
    else {
        int mid=(l+r)/2;
        add(lch,x,y,l,mid,val);
        add(rch,x,y,mid+1,r,val);
    }
}

LL sum;
void dfs(int u,int l,int r) {
    LL bak=sum;
    vector<Edge> del,add;

    for(auto [x,y,z,w]:seg[u]) {
        int rt=lct.split(x, y);
        if(lct.info(rt).maxv>w) {
            del.emplace_back(edge[lct.info(rt).maxe]);
            add.emplace_back(x,y,z,w);
            auto [a,b,c,d]=del.back();
            lct.cut(a, c);
            lct.cut(b, c);
            lct.link(x, z);
            lct.link(y, z);
            sum=sum-d+w;
        }
    }

    if(l==r) ans[l]=sum;
    else {
        int mid=(l+r)/2;
        dfs(lch, l, mid);
        dfs(rch, mid+1, r);
    }

    while(add.size()) {
        auto [x,y,z,w]=add.back();
        lct.cut(x, z);
        lct.cut(y, z);
        tie(x,y,z,w)=del.back();
        lct.link(x, z);
        lct.link(y, z);
        add.pop_back();
        del.pop_back();
    }
    sum=bak;
}

#undef lch
#undef rch

void solve() {
    int n;
    cin>>n;
```

```
    constexpr int offset=N+M;
    for(int i=1;i<n;i++) {
        int u,v,w;
        cin>>u>>v>>w;
        u+=offset,v+=offset;
        Edge edg{u,v,i,w};
        edge.emplace_back(edg);
        lct.info(i)={w,w,i,i};
        lct.link(u, i);
        lct.link(v, i);
        sum+=w;
    }
    sum++;

    int m;
    cin>>m;
    for(int i=1;i<=m;i++) {
        int u,v,w,l,r;
        cin>>u>>v>>w>>l>>r;
        u+=offset,v+=offset;
        Edge edg{u,v,i+n,w};
        edge.emplace_back(edg);

        if(u==v) continue;
        lct.info(i+n)={w,w,n+i-1,n+i-1};
        add(1, l, r, 1, D, edg);
    }

    dfs(1, 1, D);
    for(int i=1;i<=D;i++) cout<<ans[i]<<endl;
}
```

首都

题意

思路

LCT维护树重心。

根据重心性质，合并两颗树时，新重心在连接两个原重心的路径上。

2023 HDU多校 3-1001 Magma Cave

题意

给定一个 n 个点的图，刚开始没有边。q 次操作：

1. 添加无向边 (u,v)，边权 w，保证 $1 \le w \le q$ 且两两不同（也就是说 w 可以看作边的id）。
2. 询问是否存在一颗生成树，使得树中第 k 大的边边权为 w。

思路

假设生成树中已经包含边 w，考虑生成树中可能比 w 小的边的数量，可以发现这个数量实际上是一个连续的区间，区间左右端点分别在 最大/最小 生成树中取到。

简单证明：首先左右端点是毋庸置疑的，而我们必然能通过不断用大权边替换 $MinST$ 中的小权边来得到 $MaxST$，由于这个替换的过程每次最多将一条 $< w$ 的边变成 $> w$，因此可达成的 k 值必然是一个连续的区间。

因此，我们可以直接暴力开两颗 LCT 维护 $MinST, MaxST$，然后再用两颗树状数组维护对应的边权集合，用来查询 $< w$ 的边数。

另外还需要讨论的一个点就是我们需要保证 MST 中存在 w 边。

- 当 $MinST$ 中不包含 w 边时，那么说明加入 w 边所形成环中，w 必然是最大的（否则图就不满足是一个 $MinST$）。所以我们将 w 加入其中会导致 $< w$ 的边数 -1。
- 而 $MaxST$ 则不需要处理，因为不论是否将一条 $> w$ 的边变成 w，都不会导致 $< w$ 的边数发生变化。

复杂度 $\mathcal{O}(q \log q)$。

实现

> 评测记录 ~~（400多行的shit山）~~

```
// 如果边数不够或者不存在w，那么直接NO
if(cnt+1<n||!st.count(w)) io<<"NO"<<endl;
else {
    // minfen,maxfen 维护对应MST的边权集合的树状数组
    int v=minfen.query(w)-minfen.query(w-1);
    int l=maxfen.query(w-1);
    int r=minfen.query(w-1)-!v; // 如果不存在w，那么减去1
    // 算出可能比w的小的数量区间后，可行的k区间就是 [l+1,r+1]
    if(k>=l+1&&k<=r+1) io<<"YES"<<endl;
    else io<<"NO"<<endl;
}
```