Heavy-Light Decoposition 重链剖分

什么是重链剖分?

> 树链剖分
>
> 树链剖分是指，将一棵树分成若干条链，使得这些链具有某些性质，使得树上操作变为链上操作和跳链操作（链间操作）。

重链剖分（Heavy-Light Decomposition，简称 HLD）是一种把树分解为若干条链的技巧，目的是把树上的路径/子树类操作转化为若干条链上的区间操作，从而结合线段树/树状数组等数据结构高效处理。

基本概念与性质

- 选择每个节点的子树中规模最大的儿子作为它的“重儿子”，相应的边称为重边；其他边称为轻边。沿树向下，重边尽量连成长链，这些链称为重链。
- 任意节点到根路径上，至多会经过 O(log n) 条轻边（因为每次经过轻边，子树规模至少减半），这保证了分解后的链数较少。
- 对每条重链上的节点按 DFS 序（或时间戳）映射为数组的连续区间，这样路径查询可以拆成若干个区间查询；子树查询对应于以某节点为根的一段连续区间。

基本思路（实现要点）

1. 第一次 DFS（dfs1）：计算每个节点的子树大小 `size[u]`，并记录重儿子 `heavy[u]`（即 `size` 最大的儿子）。
2. 第二次 DFS（dfs2）：按重链将节点分配到链顶 `top[u]`，同时记录节点在基数组中的序号 `dfn[u]`，并把同一重链上节点映射为连续的 `dfn` 区间。
3. 将节点初值按 dfn 顺序放入线段树或树状数组，支持区间查询/区间修改。路径查询（u,v）时向上跳链：将 u、v 分别移动到它们链顶的较深端并在对应区间上查询，直到两节点在同一条链上。

复杂度：在常见实现中，单次路径查询需要跳过 O(log n) 条链，每条链上使用线段树/树状数组的操作为 O(log n)，因此总体为 O(log^2 n)。在某些优化（如链内直接维护带跳表或融合结构）下能进一步减少常数。

常见应用场景

- 路径查询与修改：例如求路径上节点权值之和/最大值、对路径上的节点加上某个值、将路径上所有权值置为某值等。
- 子树查询与修改：因为子树在 dfn 序上是连续区间，可用于子树加/子树求和/子树最大值等操作。
- 支持带权边或带权点的问题：把边权或点权映射到对应的 dfn 下（常把边权映射到子节点的 dfn 位置）。
- 结合 LCA（最近公共祖先）：在做路径分解时常用 LCA 来判断是否在同一链或结束跳链的条件。
- 动态维护与在线询问：与线段树/树状数组结合，能在线处理大量修改与查询；也可与分治、并查集等技术混用解决复杂问题。
- 竞赛题常见模式：树上区间/路径问题（例如树上 k 最近点、路径上第 k 大、树链上带约束的二分/分块处理等）。

示例说明（简短）

- 要求在树上回答若干次 "将 `u` 到 `v` 路径上的所有点加上 x" 与 "询问 `u` 到 `v` 路径上所有点的和"，使用 HLD：把树映射为基数组，路径操作拆为若干区间操作并在区间上调用线段树的区间加/区间求和。
- 要求对子树内所有节点进行更新并查询子树和：直接将子树对应的 dfn 区间作为线段树的区间操作。

代码模板

```
struct HLDT{//Heavy-Light Decomposed Tree
    struct Node{//树中的每个节点 each node in the tree
        int id;//该节点的编号 idenity
        int faid;//该节点的父节点编号 father idenity
        Node* fa;//该节点的父节点 father
        /*特别的，对于根节点 fa = nullptr faid = 0 */
        int dept;//该节点深度 depth
        int siz;//该节点的子树大小 size
        Node* hs;//该节点的重儿子 heavy_son
        int dfn;//该节点的dfs序号 DFS_number
        Node* top;//该节点所属的重链的 top 节点
        std::vector<Node*> s;//该节点的所有儿子 son
        Node(Node* u,int _id,auto& m){//新建一个编号为v的节点，指定u为他的父节点，自动添加 id 映射
            id = _id,dept = u->dept + 1,fa = u,faid = u->id,siz = 1,hs = nullptr,top = nullptr,dfn = 0;m[_id]
= this;
            u->s.push_back(this);//把该节点加入到父节点的儿子队列中
        }
        Node(auto& m){id = 0,dept = 0,fa = nullptr,faid = 0,siz = 1,hs = nullptr,top = nullptr,dfn = 0;m[id] =
this;}//新建一个编号为v的节点，指定u为他的父节点，自动添加 id 映射
    };
    Node* root;
    std::unordered_map<int,Node*> idc;//Id 控制器 id-controller
    int time = 0;//用于创建 DFN 的时间戳
    void dfs_build(Node* u){
        int heavy=0;
        for(const int v:g[u->id]){
            if(v == u->faid)continue;//不走回头路
            Node *vp = new Node(u,v,idc);//指向子节点的指针 v-pointer
            dfs_build(vp);//递归建树
            u->siz += vp->siz;//统计子树大小 count the size of subtree
            if(vp->siz > heavy)heavy = vp->siz,u->hs = vp;//选出重儿子 selection for heavy son
        }
    }
    void dfs_link(Node* u){
        u->dfn = ++time;
        if(u->s.empty())return;//叶子结点
        u->hs->top = u->top;dfs_link(u->hs);//重儿子在重链上，top 继承本节点。重儿子优先 dfs，保证重链dfn连续。
        for(const auto v:u->s){
            if(v == u->hs)continue;//重儿子已经走过
            v->top = v;dfs_link(v);//轻儿子的 top 节点是它自己
        }
    }
    void build(){dfs_build(root= new Node(idc));}//建树
    void link(){root->top = root;dfs_link(root);}//链接重链
```

```
    SGT sgt;
    HLDT(){
        build(),link();
        sgt.build(sgt.root = new SGT::Node(SGT::Rge(1,n)));
    }
};
```

此外需要一个线段树来支持区间操作，在此不赘述，见线段树章节。

树上操作

树上操作是指在树上进行的各种查询和修改操作，常见的有**路径查询、路径修改、子树查询和子树修改**等。通过重链剖分，我们可以将这些树上操作转化为**链上的区间操作**，从而利用线段树或树状数组高效地处理。

在树上的操作要被转换成在链上的操作，主要需要 LCA **指导跳链**。

LCA 指导跳链

在 HLD 中，LCA（最近公共祖先）常用于判断两个节点是否在同一条链上，以及结束跳链的条件。具体来说：

- 当查询路径 (u, v) 时，首先比较 u 和 v 的链顶节点的深度。
- 如果 u 和 v 的链顶节点不同，则将较深的节点（假设是 u）沿着链向上跳到其链顶节点的父节点，即 u = u->top->fa。
- 重复上述步骤，直到 u 和 v 的链顶节点相同，此时它们在同一条链上，可以直接在该链上进行区间查询。

> **为什么要跳链顶**
>
> 如下图所示（两条重链构成一个人字形），如果现在两节点分别处于 E、F 上，明显 E 的深度大于 F 的深度。凭感觉我们会选择较深的节点向上跳链，最终跳到同一条重链上。
>
> flowchart TB A----|"重链 1"|B B----|"重链 1"|C B----|"轻边"|D D----|"重链 2"|F C----|"重链 1"|G G----|"重链 1"|E
>
> > ✗ **以节点深度为依据**
> >
> > 我们应该跳到 A，很明显我们已经错过了 $Lca(E,F)$
>
> > ✓ **以链顶深度为依据**
> >
> > 以链顶深度为依据，我们会选择 F 向上跳链，最终跳到同一条重链上。

跳链操作

所有路径操作都可以化为跳链操作，即在跳链过程中对链上的区间进行查询或修改。下面是一个示例代码模板，展示了如何在路径 (u, v) 上进行操作：

```
void operate_on_path(int _u, int _v, std::function<void(HLDT&)> factor, HLDT &t) {
    auto u = t.f_id[_u], v = t.f_id[_v];//编号到指针

    while (u->top != v->top) {
        if (u->top->dep < v->top->dep) std::swap(u, v);
        t.sgt.inrg = HLDT::SGT::Rge(u->top->dfn, u->dfn);
        factor(t);
        u = u->top->fa;
    }

    if (u != v) {
        if (u->dep < v->dep) std::swap(u, v);
        t.sgt.inrg = HLDT::SGT::Rge(v->dfn + 1, u->dfn);
        factor(t);
    }
}
```

或者写成这样。

```
void operate(int _u,int _v,std::function<void()> factor){
    auto u = f_id[_u],v = f_id[_v];
    for(;u->top != v->top;u = u->top->fa){
        if(u->top->dep < v->top->dep)std::swap(u,v);
        sgt.inrg = SGT::Rge(u->top->dfn,u->dfn);
        factor();
    }
    sgt.inrg = SGT::Rge(u->dfn,v->dfn);
    factor();
}
```

其中 `std::function<void(HLDT&)> factor` 是一个函数对象，表示在链上进行的具体操作（例如区间加、区间求和等）。。`factor` 要写成lambda表达式的形式，方便在调用时传入具体的操作逻辑。

以路径加为例：

```
void Add(int _u, int _v, int w, HLDT &t) {
    operate_on_path(_u, _v, [w](HLDT &tree) {
        tree.sgt.modify(tree.sgt.root, [w](auto node) {
            node->update_by(w);
        });
    }, t);
}
```

例题

Luogu P3384 【模板】树链剖分

本题是树链剖分的模板题，主要考察对概念的理解和代码实现能力。题目要求在树上进行路径查询和修改，适合用 HLD 来解决。

Luogu P2146 【NOI 2015】软件包管理器

在树上进行大规模的操作，适合使用 HLD 解决。题目要求输出影响的软件包数量，可以想到把每个节点的状态映射到 HLD 的基数组上，通过查询**全局和**来统计受影响的软件包数量。

参考代码

Luogu P4315 月下毛景树

本题有所不同的地方在于，我们通常操作的是某个节点的点权（点状态）。但是本题中**有权的是边而不是点**，我们需要将边权转化为点权来使用 HLD。

常见的做法是把边权映射到子节点的点权上（因为一个节点有且仅有一个父节点，有**一一对应的关系**，借助这种关系，让某个点的**点权**表示与其父节点连边的**边权**），这样在进行路径查询或修改时，就可以直接在对应的区间上操作。

> **特别注意**
>
> 这样转换点权之后，两个节点的 LCA 的点权需要排除。因为 LCA 的点权其实是 LCA 与其父节点的边的边权，并不在 $u \to v$ 的路径上。

参考代码