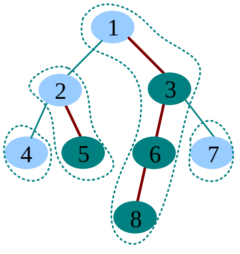
4.3　树链剖分

原理　树链剖分详解

链剖分，指对树的边进行划分的一类操作，目的是减少在链上修改、查询等操作的复杂度。链剖分有三类：轻重链剖分、虚实链剖分和长链剖分。

树链剖分的思想是通过轻重链剖分将树分为多条链，保证每个节点都属于且只属于一条链。树链剖分是轻重链剖分，节点到重儿子（子树节点数最多的儿子）之间的路径为重链。每条重链都相当于一段区间，把所有重链首尾相接组成一个线性节点序列，再通过数据结构（如树状数组、SBT、伸展树、线段树等）来维护即可。

若 size[u]表示以 u 为根的子树的节点个数，则在 u 的所有儿子中，size 最大的儿子就是重儿子，而 u 的其他儿子都是轻儿子，当前节点与其重儿子之间的边就是重边，多条重边相连为一条重链。一棵树如下图所示。长度大于 1 的重链有两条：1-3-6-8、2-5，单个轻儿子可被视作一个长度为 1 的重链：4、7，因此本题中有 4 条重链。图中深色的节点是重儿子，加粗的边是重边。



重要性质：

- 若 v 是轻儿子，u 是 v 的父节点，则 size[v]≤size[u]/2；
- 从根到某一点路径上，不超过 $\log_2 n$ 条重链，不超过 $\log_2 n$ 条轻边。

树链剖分支持以下操作。

（1）单点修改：修改一个点的权值。

（2）区间修改：修改节点 u 到 v 路径上节点的权值。

（3）区间最值查询：查询节点 u 到 v 路径上节点的最值。

（4）区间和查询：查询节点 u 到 v 路径上节点的和值。

树链剖分的应用比倍增更广泛，倍增可以做的，树链剖分一定可以做，反过来则不行。树链剖分的代码复杂度不算特别高，调试也不难，树链剖分在算法竞赛中是必备知识。

1．预处理

树链剖分可以采用两次深度优先搜索实现。

第 1 次深度优先搜索维护 4 个信息：dep[]、fa[]、size[]、son[]。

- dep[u]：u 的深度。
- fa[u]：u 的父节点。
- size[u]：以 u 为根的子树的节点数。
- son[u]：u 的重儿子，u–son[u]为重边。

第 2 次深度优先搜索以优先走重边的原则，维护 3 个信息：top[]、id[]、rev[]。

- top[u]：u 所在的重链上的顶端节点编号（重链上深度最小的节点）。
- id[u]：u 在节点序列中的位置下标。
- rev[x]：树链剖分后节点序列中第 x 个位置的节点。

id[]与 rev[]是互逆的。例如，节点 u 在节点序列中的位置下标是 x，则节点序列中第 x 个位置的节点是 u，id[u]=x，rev[x]=u。对上面的树进行树链剖分后，将所有重链都放在一起组成一个节点序列：[1,3,6,8],[7],[2,5],[4]。序列中第 4 个位置是 8 号节点，8 号节点的存储下标是 4，即 rev[4]=8，id[8]=4。预处理的时间复杂度为 $O(n)$。

2．求解 LCA 问题

对于 LCA（最近公共祖先）问题，点和边均没有权值，因此无须维护线段树来实现。输入树后，先进行树链剖分预处理。

算法代码：

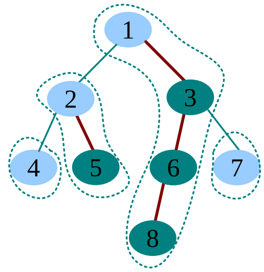
```
void dfs1(int u,int f) {//求dep、fa、size和son
    size[u]=1;
    for(int i=head[u];i;i=e[i].next){
        int v=e[i].to;
        if(v==f)//父节点
            continue;
        dep[v]=dep[u]+1;//深度
        fa[v]=u;
        dfs1(v,u);
        size[u]+=size[v];
        if(size[v]>size[son[u]])
            son[u]=v;
    }
}

void dfs2(int u) {//求top
    if(u==son[fa[u]])
```

```
      top[u]=top[fa[u]];
   else
      top[u]=u;
   for(int i=head[u];i;i=e[i].next){
      int v=e[i].to;
      if(v!=fa[u])
         dfs2(v);
   }
}
```

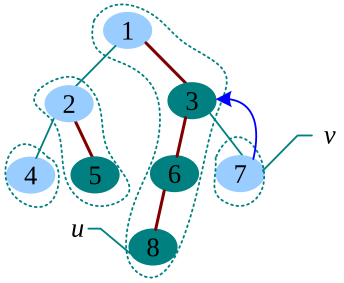
显然，树中的任意一对节点(u,v)只存在两种情况：①在同一条重链上（top[u]=top[v]）；②不在同一条重链上。

对第 1 种情况，LCA(u,v)就是 u、v 中深度较小的节点。例如下图中求节点 3 和 8 的最近公共祖先时，因为 3 和 8 在同一条重链上且 3 的深度较小，因此 LCA(3,8)=3。



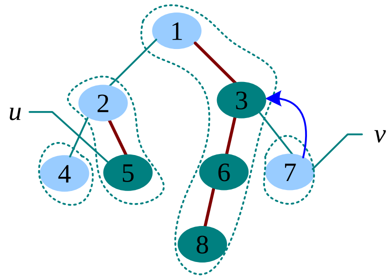
对第 2 种情况，只要想办法将 u、v 两点转移到同一条重链上即可。首先求出 u、v 所在重链的顶端节点 top[u]和 top[v]，将其顶端节点深度大的节点上移，直到 u、v 在同一条重链上，再用第 1 种情况中的方法求解即可。

例如下图中求节点 7 和 8 的最近公共祖先，7 和 8 不在同一条重链上，先求两个节点所在重链的顶端节点：top[7]=7，top[8]=1，dep[1]<dep[7]，7 的顶端节点深度大，因此将 v 从 7 上移到其父节点 3，此时 3 和 8 在同一条重链上，且 3 的深度较小，因此 LCA(7,8)=3。

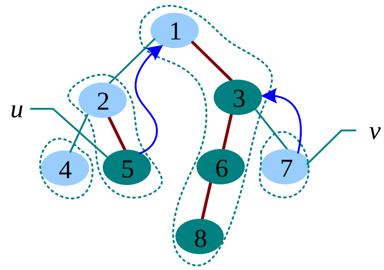


求 5 和 7 的最近公共祖先，5 和 7 不在同一条重链上，先求两节点所在重链的顶端节点：

top[5]=2，top[7]=7，dep[2]<dep[7]，7 的顶端节点深度大，因此将 *v* 从 7 上移到其顶端节点的父节点 3。



3 所在重链的顶端节点：top[3]=1，dep[1]<dep[2]，5 的顶端节点深度大，因此将 *u* 从 5 上移到其顶端节点的父节点 1，此时 1 和 3 在同一条重链上，且 1 的深度较小，因此 LCA(5,7)=1。



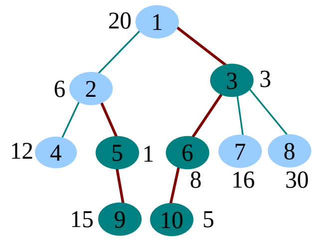
算法代码：

```
int LCA(int u,int v) {//求区间 u、v 的最近公共祖先
    while(top[u]!=top[v]) {//不在同一条重链上
        if(dep[top[u]]>dep[top[v]])//将顶端节点深度大的上移
            u=fa[top[u]];
        else
            v=fa[top[v]];
    }
    return dep[u]>dep[v]?v:u;//返回深度小的节点
}
```

3. 树链剖分与线段树

若在树中进行点更新、区间更新、区间查询等操作，则可以使用线段树来维护和处理。

一棵树如下图所示。



树链剖分之后的节点序列和下标序列如下图所示。

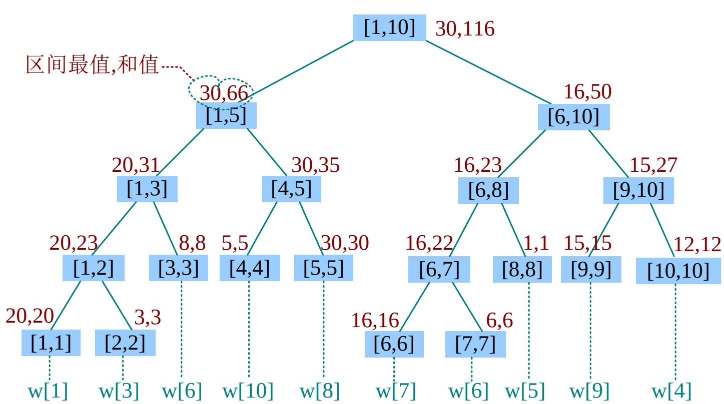
	1	2	3	4	5	6	7	8	9	10
rev[]	1	3	6	10	8	7	2	5	9	4

	1	2	3	4	5	6	7	8	9	10
id[]	1	7	2	10	8	3	6	5	9	4

节点序列对应的权值如下图所示。

	1	3	6	10	8	7	2	5	9	4
w[]	20	3	8	5	30	16	6	1	15	12

根据 w[]序列创建线段树，如下图所示。



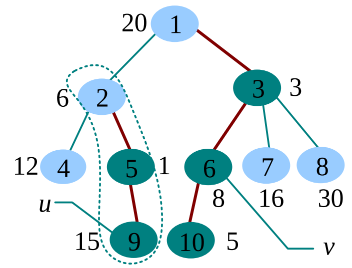
查询节点 *u* 到 *v* 路径上节点权值的最值与和值的方法如下。

- 若 *u* 和 *v* 在同一条重链上，则在线段树上查询其对应的下标区间[id[*u*],id[*v*]]即可。
- 若 *u* 和 *v* 不在同一条重链上，则一边查询，一边将 *u* 和 *v* 向同一条重链上移，然后采用上面的方法处理。对于顶端节点深度大的节点，先查询其到顶端节点的区间，然后一边上移一边查询，直到上移到同一条重链上，再查询在同一条重链上的区间。

查询节点 6～9 权值的最值与和值（包括 6 和 9 节点），过程如下。

（1）读取 top[6]=1，top[9]=2，两者不相等则说明其不在一条重链上，且 top[9]的深度大，

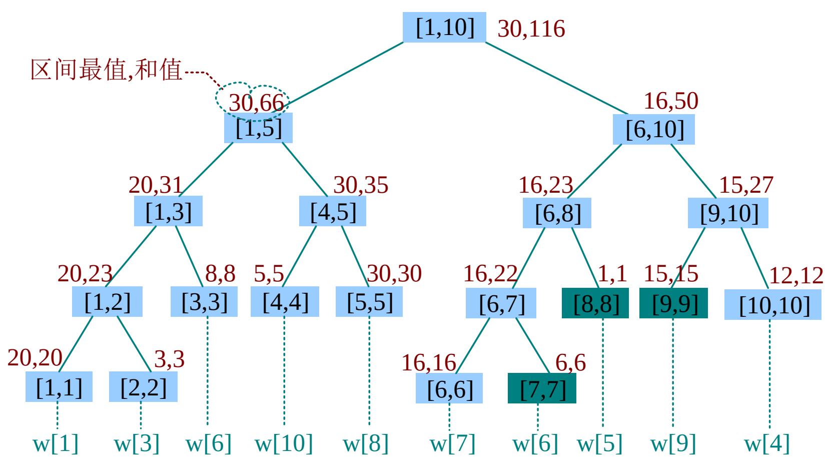
先查询 top[9]～9 之间的最值与和值。



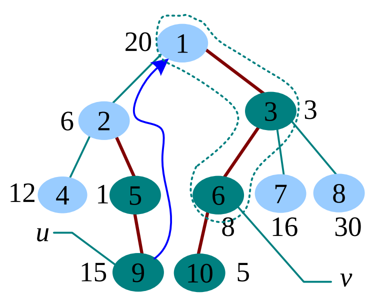
首先得到节点 2 和 9 对应的节点序列下标 7 和 9。

	1	2	3	4	5	6	7	8	9	10
id[]	1	7	2	10	8	3	6	5	9	4

然后在线段树中查询[7,9]区间的最值与和值。[7,9]区间的最值与和值：Max=15，Sum=22。



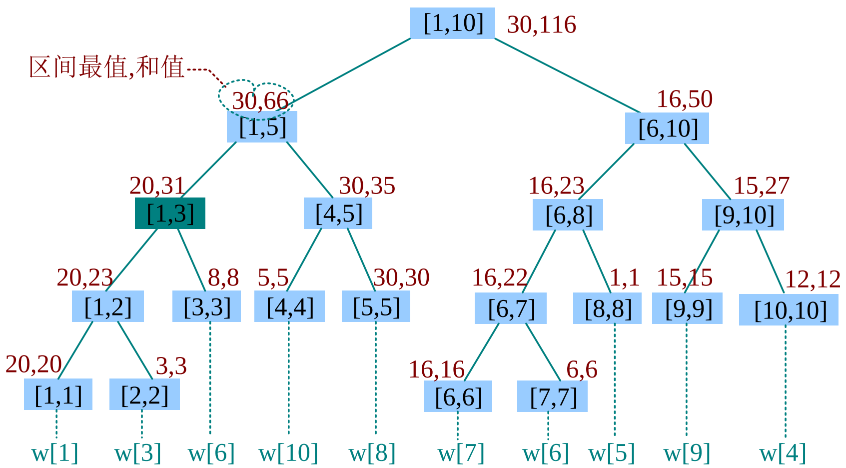
（2）将 u 上移到 top[9]（2 号节点）的父节点，即 1 号节点，此时 1 和 6 在同一条链上。



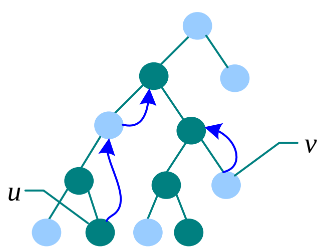
节点 1 和 6 对应的线段树下标为 1 和 3。

	1	2	3	4	5	6	7	8	9	10
id[]	1	7	2	10	8	3	6	5	9	4

在线段树中查询到[1,3]区间的最值与和值分别为 20、31，如下图所示。再与前面的结果求最大值与和值，则 Max=max(Max,20)=max(15,20)=20，Sum=Sum+31=22+31=53。



区间更新的方法与此类似，若不在一条链上，则一边更新，一边向同一条链上靠，最后在同一条链上更新即可。



注意：更新和查询时均需要先得到节点对应的线段树下标，再在线段树上更新和查询。

算法代码：

```
void dfs1(int u,int f) {//求 dep、fa、size、son
    size[u]=1;
    for(int i=head[u];i;i=e[i].next){
        int v=e[i].to;
        if(v==f)//父节点
```

```
                continue;
            dep[v]=dep[u]+1;//深度
            fa[v]=u;
            dfs1(v,u);
            size[u]+=size[v];
            if(size[v]>size[son[u]])
                son[u]=v;
        }
}

void dfs2(int u,int t){//求 top、id、rev
        top[u]=t;
        id[u]=++total;  //u 对应的节点序列中的下标
        rev[total]=u;   //节点序列下标对应的节点 u
        if(!son[u])
            return;
        dfs2(son[u],t);//沿着重儿子深度优先搜索
        for(int i=head[u];i;i=e[i].next){
            int v=e[i].to;
            if(v!=fa[u]&&v!=son[u])
                dfs2(v,v);
        }
}

void build(int k,int l,int r){//创建线段树，k 表示存储下标，区间为[l,r]
        tree[k].l=l;
        tree[k].r=r;
        if(l==r){
            tree[k].mx=tree[k].sum=w[rev[l]];
            return;
        }
        int mid,lc,rc;
        mid=(l+r)/2;//划分点
        lc=k*2;  //k 节点的左子节点存储下标
        rc=k*2+1;//k 节点的右子节点存储下标
        build(lc,l,mid);
        build(rc,mid+1,r);
        tree[k].mx=max(tree[lc].mx,tree[rc].mx);//节点的最大值等于左右子节点最值的最大值
        tree[k].sum=tree[lc].sum+tree[rc].sum;//节点的和值等于左右子树的和值
}

void query(int k,int l,int r){//求[l,r]区间的最值、和值
        if(tree[k].l>=l&&tree[k].r<=r) {//找到该区间
            Max=max(Max,tree[k].mx);
```

```
        Sum+=tree[k].sum;
        return;
    }
    int mid,lc,rc;
    mid=(tree[k].l+tree[k].r)/2;//划分点
    lc=k*2;  //左子节点存储下标
    rc=k*2+1;//右子节点存储下标
    if(l<=mid)
        query(lc,l,r);//到左子树中查询
    if(r>mid)
        query(rc,l,r);//到右子树中查询
}

void ask(int u,int v){//求 u、v 之间的最值或和值
    while(top[u]!=top[v]) {//不在同一条重链上
        if(dep[top[u]]<dep[top[v]])
            swap(u,v);
        query(1,id[top[u]],id[u]);//u 顶端节点和 u 之间
        u=fa[top[u]];
    }
    if(dep[u]>dep[v]) //在同一条重链上
        swap(u,v);    //深度小的节点为 u
    query(1,id[u],id[v]);
}

void update(int k,int i,int val){//u 对应的下标 i，将其值更新为 val
    if(tree[k].l==tree[k].r&&tree[k].l==i){//找到 i
        tree[k].mx=tree[k].sum=val;
        return;
    }
    int mid,lc,rc;
    mid=(tree[k].l+tree[k].r)/2;//划分点
    lc=k*2;  //左子节点存储下标
    rc=k*2+1;//右子节点存储下标
    if(i<=mid)
        update(lc,i,val);//到左子树中更新
    else
        update(rc,i,val);//到右子树中更新
    tree[k].mx=max(tree[lc].mx,tree[rc].mx);//返回时更新最值
    tree[k].sum=tree[lc].sum+tree[rc].sum;//返回时更新和值
}
```

算法分析：树链剖分预处理需要 $O(n)$时间，每次更新和查询都需要 $O(\log n)$时间。

训练 1　树上距离

题目描述（HDU2586）见 2.2 节训练 2。

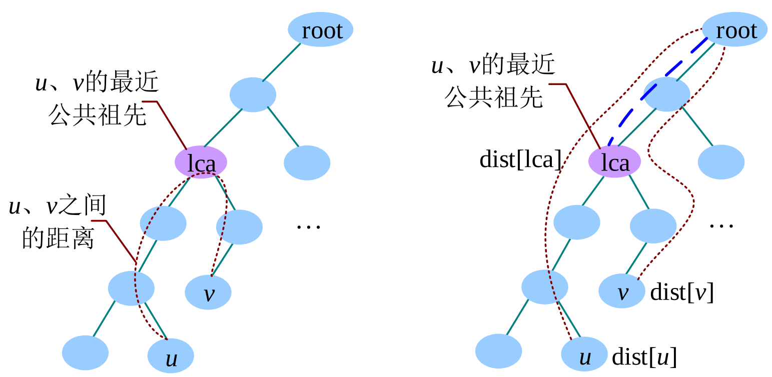
题解：由于本题中任意两个房子之间的路径都是唯一的，找它们之间的距离等价于在树中找两个节点之间的距离，所以可以采用最近公共祖先 LCA 的方法求解。求解 LCA 的方法很多，例如树上倍增+ST，在此采用树链剖分解决。

1. 算法设计

（1）采用树链剖分将树转变为线性序列。

（2）求两个节点的最近公共祖先。

（3）求 *u* 和 *v* 之间的距离。若 *u* 和 *v* 的最近公共祖先为 lca，则 *u* 和 *v* 之间的距离为 *u* 到树根的距离加上 *v* 到树根的距离，减去 2 倍的 lca 到树根的距离：dist[*u*]+dist[*v*] –2 ×dist[lca]。



2. 算法实现

```
void dfs1(int u,int f){//求dep、fa、size、son、dist
	size[u]=1;
	for(int i=head[u];i;i=e[i].next){
		int v=e[i].to;
		if(v==f)//父节点
			continue;
		dep[v]=dep[u]+1;//深度
		fa[v]=u;
		dist[v]=dist[u]+e[i].c;//距离
		dfs1(v,u);
		size[u]+=size[v];
		if(size[v]>size[son[u]])
			son[u]=v;
	}
}
```

```
void dfs2(int u){//求top
     if(u==son[fa[u]])
          top[u]=top[fa[u]];
     else
          top[u]=u;
     for(int i=head[u];i;i=e[i].next){
          int v=e[i].to;
          if(v!=fa[u])
               dfs2(v);
     }
}

int LCA(int u,int v){//求u、v的最近公共祖先
     while(top[u]!=top[v]){//不在同一条重链上
          if(dep[top[u]]>dep[top[v]])
               u=fa[top[u]];
          else
               v=fa[top[v]];
     }
     return dep[u]>dep[v]?v:u;//返回深度小的节点
}

for(int i=1;i<=m;i++){
     cin>>x>>y;
     lca=LCA(x,y);
     cout<<dist[x]+dist[y]-2*dist[lca]<<endl;//输出x、y的距离
}
```

训练 2　树的统计

题目描述（HYSBZ1036）：一棵树有 n 个节点，编号为 1～n，每个节点都有一个权值 w。完成以下操作：①CHANGE u t，把节点 u 的权值修改为 t；②QMAX u v，询问从节点 u 到节点 v 路径上节点的最大权值；③QSUM u v，询问从节点 u 到节点 v 路径上节点的权值和（注意：从节点 u 到节点 v 路径上的节点包括 u 和 v 自身）。

输入：第 1 行包含一个整数 n，表示节点的个数。接下来的 n–1 行，每行都包含两个整数 a 和 b，表示在节点 a 和节点 b 之间有一条边相连。接下来的 n 行，第 i 行的整数 w_i 表示节点 i 的权值。接下来的一行包含一个整数 q，表示操作的总数。最后有 q 行，每行都表示一种操作，操作形式如上所述。其中，1≤n≤30000，0≤q≤200000，保证操作中每个节点的权值 w 都为–30000～30000。

输出：对每个 QMAX 或者 QSUM 的操作，都单行输出一个整数表示要求的结果。

输入样例	输出样例

<pre>4
1 2
2 3
4 1
4 2 1 3
12
QMAX 3 4
QMAX 3 3
QMAX 3 2
QMAX 2 3
QSUM 3 4
QSUM 2 1
CHANGE 1 5
QMAX 3 4
CHANGE 3 6
QMAX 3 4
QMAX 2 4
QSUM 3 4</pre>
	<pre>4
	1
	2
	2
	10
	6
	5
	6
	5
	16</pre>

题解：本题包括树上点更新、区间最值、区间和值查询。可以用树链剖分将树形结构线性化，然后用线段树进行点更新、区间最值、区间和值查询。解决方案：树链剖分+线段树。

算法设计：

（1）第 1 次深度优先遍历求 dep、fa、size、son，第 2 次深度优先遍历求 top、id、rev；

（2）创建线段树；

（3）点更新，u 对应的下标 i=id[u]，在线段树中将该下标的值更新为 val；

（4）区间查询，求 u、v 之间的最值与和值。若 u、v 不在同一条重链上，则一边查询，一边向同一条重链靠拢；若 u、v 在同一条重链上，则根据节点的下标在线段树中进行区间查询。

算法实现源码见下载文件。

训练 3　家庭主妇

题目描述（POJ2763）：X 村的人们住在美丽的小屋里。若两个小屋通过双向道路连接，则可以说这两个小屋直接相连。X 村非常特别，可以从任意小屋到达任意其他小屋，每两个小屋之间的路线都是唯一的。温迪的子节点喜欢去找其他子节点玩，然后打电话给温迪："妈咪，带我回家！"。在不同的时间沿道路行走所需的时间可能不同。温迪想告诉她的子节点她将在路上花的确切时间。

输入：第 1 行包含 3 个整数 n、q、s，表示有 n 个小屋、q 个消息，温迪目前在 s 小屋里，n<100001，q<100001。以下 n–1 行各包含 3 个整数 a、b 和 w，表示有一条连接小屋 a 和 b 的道

路，所需的时间是 w（$1\leqslant w\leqslant 10000$）。以下 q 行有两种消息类型：①消息 A，即 0 u，子节点在小屋 u 中给温迪打电话，温迪应该从现在的位置去小屋 u；②消息 B，即 1 i w，将第 i 条道路所需的时间修改为 w（注意：温迪在途中时，时间不会发生改变，时间在温迪停留在某个地方等待子节点时才会改变）。

输出：对每条消息 A，都输出一个整数，即找到子节点所需的时间。

输入样例	输出样例
3 3 1 1 2 1 2 3 2 0 2 1 2 3 0 3	1 3

题解：本题中任意两个小屋都可以相互到达，且路径唯一，明显是树形结构。可以将边权看作点权，对一条边，让深度 dep 较大的点存储边权。对边 u、v，边权为 w，若 dep[u]>dep[v]，则视 u 的权值为 w。本题包括树上点更新、区间和查询。可以用树链剖分将树形结构线性化，然后用线段树进行点更新、区间和查询。解决方案：树链剖分+线段树。

算法设计：

（1）第 1 次深度优先遍历求 dep、fa、size、son，第 2 次深度优先遍历求 top、id、rev；

（2）创建线段树；

（3）点更新，u 对应的下标 i=id[u]，将其值更新为 val；

（4）区间查询，求 u、v 之间的和值。若 u、v 不在同一条重链上，则一边查询，一边向同一条重链靠拢；若 u、v 在同一条重链上，则根据节点的下标在线段树中进行区间查询。注意：因为在本题中是将边权转变为点权，所以实际查询的区间应为 query(1,id[son[u]],id[v])。

算法实现源码见下载文件。

训练 4　树上操作

题目描述（POJ3237）：一棵树的节点编号为 1～N，边的编号为 1～N–1，每条边都带有权值。在树上执行一系列指令，形式如下。

CHANGE i v	将第 i 条边的权值更改为 v
NEGATE a b	将点 a 到 b 路径上每条边的权值都改为其相反数
QUERY a b	找出点 a 到 b 路径上边的最大权值

输入：输入包含多个测试用例。第 1 行为测试用例的数量 T（$T\leqslant 20$）。每个测试用例的前面都有一个空行。第 1 个非空行包含 N（$N\leqslant 10,000$）。接下来的 N–1 行，每行都包含 3 个整数

a、*b* 和 *c*，表示边的两个节点 *a* 和 *b* 及该边的权值 *c*。边按输入的顺序编号。若在行中出现单词“DONE”，则标志着结束。

输出：对每条 QUERY 指令，都单行输出结果。

输入样例	输出样例
1 3 1 2 1 2 3 2 QUERY 1 2 CHANGE 1 3 QUERY 1 2 DONE	1 3

题解：在本题中可以将边权看作点权，对一条边，让深度 dep 较大的节点存储边权，例如边 *u*、*v*，边权为 *w*，若 dep[*u*]>dep[*v*]，则 *u* 的权值为 *w*。本题涉及树上点更新、区间更新、区间最值查询，可以用树链剖分将树形结构线性化，然后用线段树进行点更新、区间更新、区间最值查询。解决方案：树链剖分+线段树。

1．算法设计

（1）第 1 次深度优先遍历求 dep、fa、size、son，第 2 次深度优先遍历求 top、id、rev。

（2）创建线段树。

（3）点更新，*u* 对应的下标 *i*=id[*u*]，将其值更新为 val。

（4）区间查询，求 *u*、*v* 之间的最大值。若 *u*、*v* 不在同一条重链上，则一边查询，一边向同一条重链靠拢；若 *u*、*v* 在同一条重链上，则根据节点的下标在线段树中进行区间查询。注意：因为本题是将边权转变为点权，所以实际查询的区间应为 query(1,id[son[*u*]],id[*v*])。

（5）区间更新，把 *u*、*v* 路径上边的值变为相反数。像区间查询一样，需要判断 *u*、*v* 是否在一条重链上分别处理。取反后更新最大值，可以将最大值和最小值取反后交换，并打懒标记。

2．算法实现

```
void dfs1(int u,int f){}//求dep、fa、size、son
void dfs2(int u,int t){}//求top、id、rev
void build(int i,int l,int r){}//创建线段树，i表示存储下标，[l,r]为区间
void push_up(int i){//上传
    tree[i].Max=max(tree[i<<1].Max,tree[(i<<1)|1].Max);
    tree[i].Min=min(tree[i<<1].Min,tree[(i<<1)|1].Min);
}

void push_down(int i){//下传
```

```
    if(tree[i].l==tree[i].r) return;
    if(tree[i].lazy){//下传给左右子节点，懒标记清零
        tree[i<<1].Max=-tree[i<<1].Max;
        tree[i<<1].Min=-tree[i<<1].Min;
        swap(tree[i<<1].Min,tree[i<<1].Max);
        tree[(i<<1)|1].Max=-tree[(i<<1)|1].Max;
        tree[(i<<1)|1].Min=-tree[(i<<1)|1].Min;
        swap(tree[(i<<1)|1].Max,tree[(i<<1)|1].Min);
        tree[i<<1].lazy^=1;
        tree[(i<<1)|1].lazy^=1;
        tree[i].lazy=0;
    }
}

void update(int i,int k,int val){//点更新，线段树的第 k 个值为 val
    if(tree[i].l==k&&tree[i].r==k){
        tree[i].Max=val;
        tree[i].Min=val;
        tree[i].lazy=0;
        return;
    }
    push_down(i);
    int mid=(tree[i].l+tree[i].r)/2;
    if(k<=mid) update(i<<1,k,val);
    else update((i<<1)|1,k,val);
    push_up(i);
}

void update2(int i,int l,int r){//区间更新，线段树[l,r]区间的权值变为相反数
    if(tree[i].l>=l&&tree[i].r<=r){
        tree[i].Max=-tree[i].Max;
        tree[i].Min=-tree[i].Min;
        swap(tree[i].Max,tree[i].Min);
        tree[i].lazy^=1;
        return;
    }
    push_down(i);
    int mid=(tree[i].l+tree[i].r)/2;
    if(l<=mid) update2(i<<1,l,r);
    if(r>mid) update2((i<<1)|1,l,r);
    push_up(i);
}

void query(int i,int l,int r){//查询线段树中[l,r]区间的最大值
```

```
    if(tree[i].l>=l&&tree[i].r<=r){//找到该区间
        Max=max(Max,tree[i].Max);
        return;
    }
    push_down(i);
    int mid=(tree[i].l+tree[i].r)/2;
    if(l<=mid) query(i<<1,l,r);
    if(r>mid) query((i<<1)|1,l,r);
    push_up(i);
}

void ask(int u,int v){//求u、v之间的最值
    while(top[u]!=top[v]){//不在同一条重链上
        if(dep[top[u]]<dep[top[v]])
            swap(u,v);
        query(1,id[top[u]],id[u]);//u顶端节点和u之间
        u=fa[top[u]];
    }
    if(u==v) return;
    if(dep[u]>dep[v])//在同一条重链上
        swap(u,v); //深度小的节点为u
    query(1,id[son[u]],id[v]);//注意：是son[u]
}

void Negate(int u,int v){//把u-v路径上边的值变为相反数
    while(top[u]!=top[v]){//不在同一条重链上
        if(dep[top[u]]<dep[top[v]])
            swap(u,v);
        update2(1,id[top[u]],id[u]);//u顶端节点和u之间
        u=fa[top[u]];
    }
    if(u==v) return;
    if(dep[u]>dep[v])//在同一条重链上
        swap(u,v); //深度小的节点为u
    update2(1,id[son[u]],id[v]);//注意：是son[u]
}
```