实连剖分——LCT

主讲：程子奇

感谢dxw&wzd的博客给予我灵感！

同样将某一个儿子的连边划分为实边，而连向其他子树的边划分为虚边。
区别在于虚实是可以动态变化的，因此要使用更高级、更灵活的Splay来维护每一条由若干实边连接而成的实链。
基于性质更加优秀的实链剖分，LCT(Link-Cut Tree)应运而生。
LCT维护的对象其实是一个森林。

在实链剖分的基础下，LCT资磁更多的操作

1）查询、修改链上的信息（最值，总和等）
2）随意指定原树的根（即换根）
3）动态连边、删边
4）合并两棵树、分离一棵树
5）动态维护连通性
6）更多意想不到的操作 Coming Soon……

LCT和静态的树链剖分很像。怎么说呢？这两种树形结构都是由若干条长度不等的“重链”和“轻边”构成 “重链”之间由”轻边”连接。
LCT和树链剖分不同的是，树链剖分的链是不会变化的，所以可以很方便的用线段树维护。但是,既然是动态树，那么树的结构形态将会发生改变，所以我们要用更加灵活的维护区间的结构来对链进行维护,不难想到Splay可以胜任。

- 在这里解释一下为什么要把树砍成一条条的链：我们可以在logn的时间内维护长度为n的区间（链），所以这样可以极大的提高树上操作的时间效率。在树链剖分中，我们把一条条链放到线段树上维护。但是LCT中，由于树的形态变化，所以用能够支持合并、分离、翻转等操作的Splay维护LCT的重链（注意，单独一个节点也算是一条重链）。

- 这时我们注意到，LCT中的轻边信息变得无法维护。为什么呢？因为Splay只维护了重链，没有维护重链之间的轻边；而LCT中甚至连根都可以不停的变化，所以也没法用点权表示它父边的边权（父亲在变化）。所以，如果在LCT中要维护边上信息，个人认为最方便的方法应该是把边变成一个新点和两条边。这样可以把边权的信息变成点权维护，同时为了不影响，把真正的树上节点的点权变成0，就可以用维护点的方式维护边。

LCT的主要性质如下：

性质1：每一个Splay维护的是一条从上到下按在原树中深度严格递增的路径，且中序遍历Splay得到的每个点的深度序列严格递增。

比如有一棵树，根节点为1（深度1），有两个儿子2,3（深度2），那么Splay有3种构成方式：

{1-2},{3}

{1-3},{2}

{1},{2},{3}

（每个集合表示一个Splay）

而不能把1,2,3同放在一个Splay中（存在深度相同的点）

性质2：每个节点包含且仅包含于一个Splay中

性质3：边分为实边和虚边，实边包含在Splay中，而虚边总是由一棵Splay指向另一个节点（指向该Splay中中序遍历最靠前的点在原树中的父亲）。

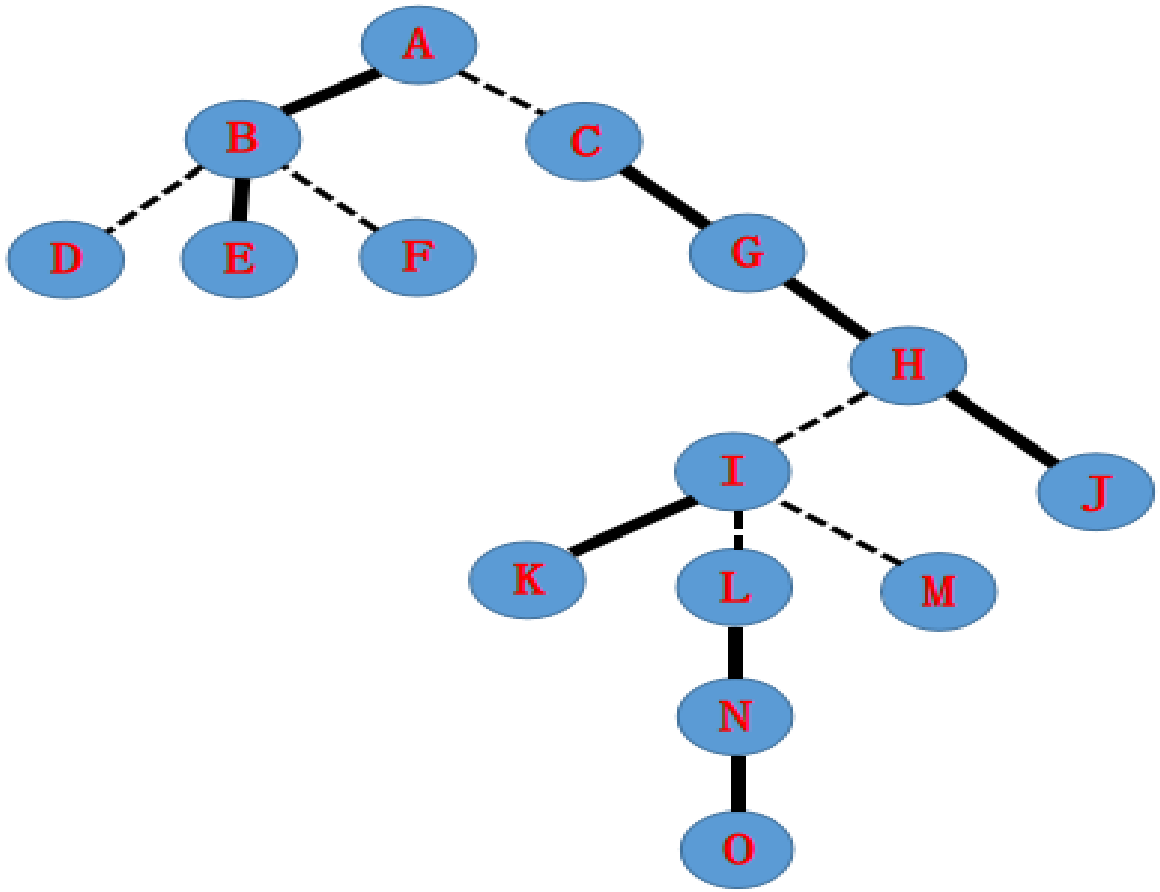
因为性质2，当某点在原树中有多个儿子时，只能向其中一个儿子拉一条实链（只认一个儿子），而其它儿子是不能在这个Splay中的。

那么为了保持树的形状，我们要让到其它儿子的边变为虚边，由对应儿子所属的Splay的根节点的父亲指向该点，而从该点并不能直接访问该儿子（认父不认子）。

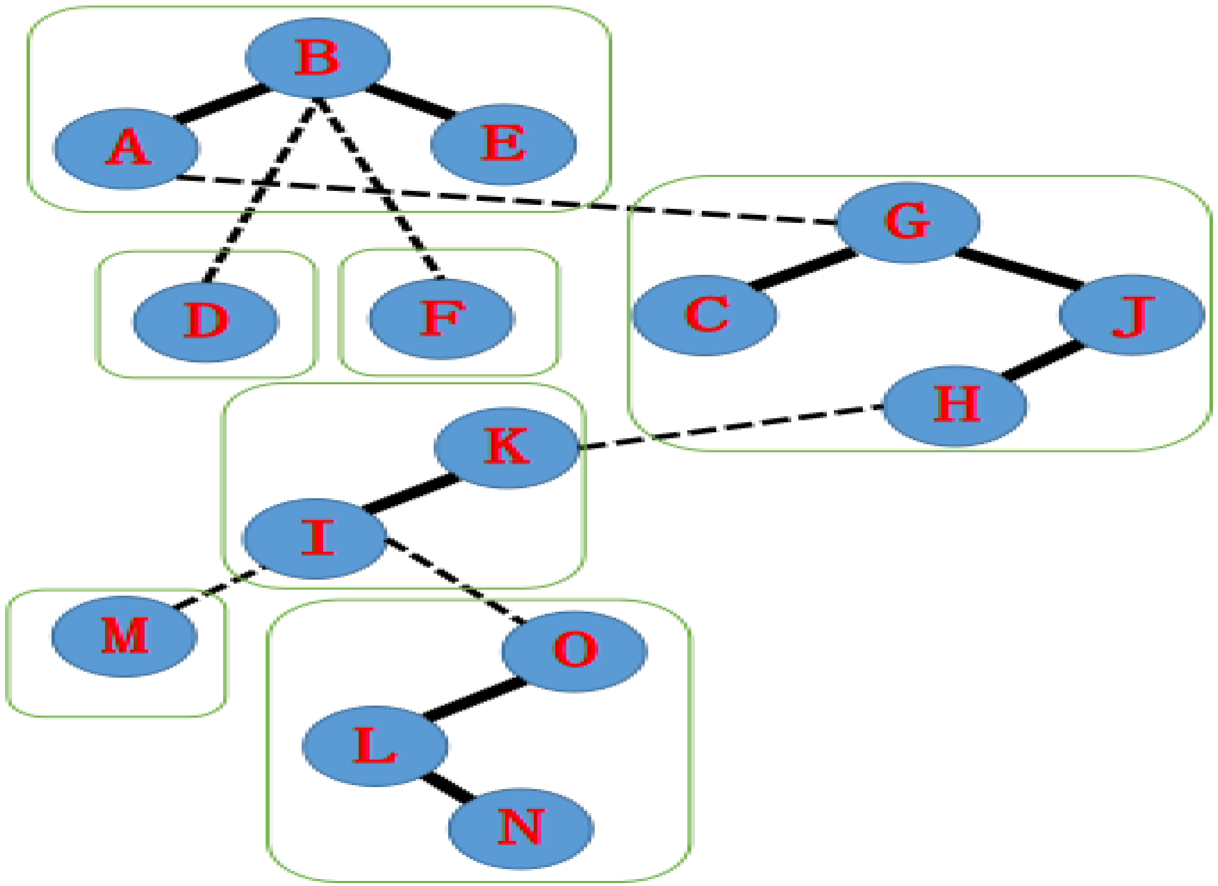
LCT核心操作:access

因为性质3，我们不能总是保证两个点之间的路径是直接连通的（在一个Splay上）。access即定义为**打通根节点到指定节点的实链**，使得一条中序遍历以根开始、以指定点结束的Splay出现。

栗子：有一棵树，假设一开始实边和虚边是这样划分的（虚线为虚边）



那么所构成的LCT可能会长这样（绿框中为一个Splay，可能不会长这样，但只要满足中序遍历按深度递增（性质1）就对结果无影响）

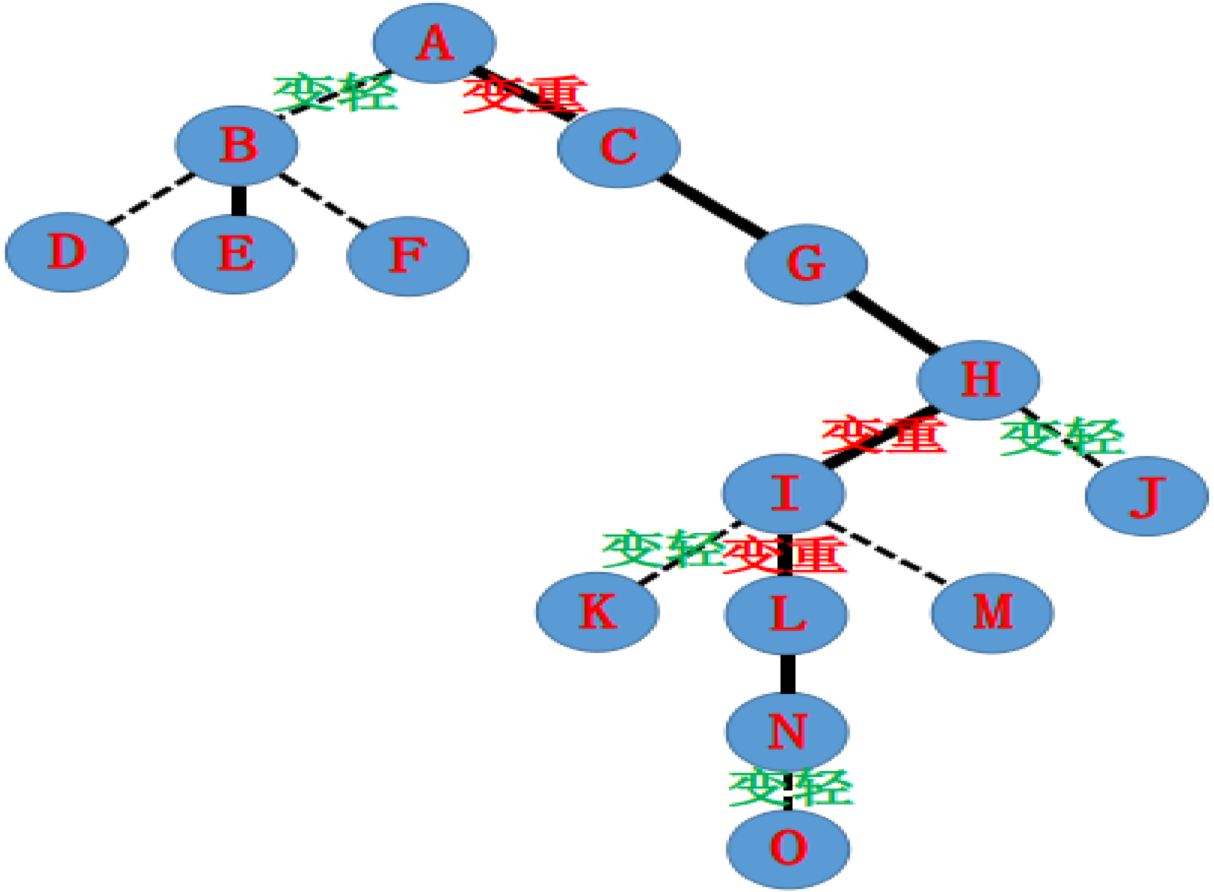


LCT核心操作:access

现在我们要access(N)，把A-N的路径拉起来变成一条Splay。

因为性质2，该路径上其它链都要给这条链让路，也就是把每个点到该路径以外的实边变虚。

所以我们希望虚实边重新划分成这样。



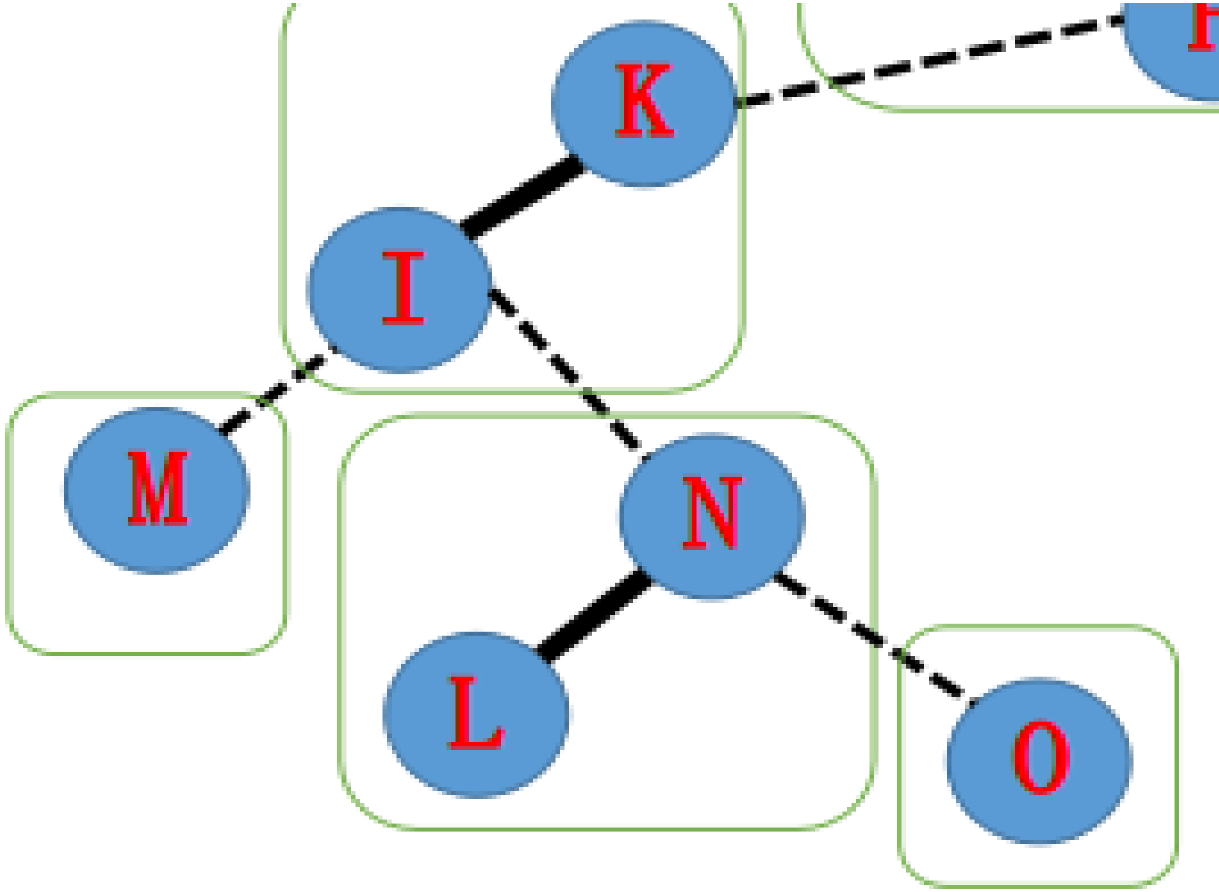
我们要一步步往上拉。

首先把splay(N)，使之成为当前Splay中的根。

为了满足性质2，原来N-O的重边要变轻。

因为按深度O在N的下面，在Splay中O在N的右子树中，所以直接单方面将N的右儿子置为0（认父不认子）

然后就变成了这样——



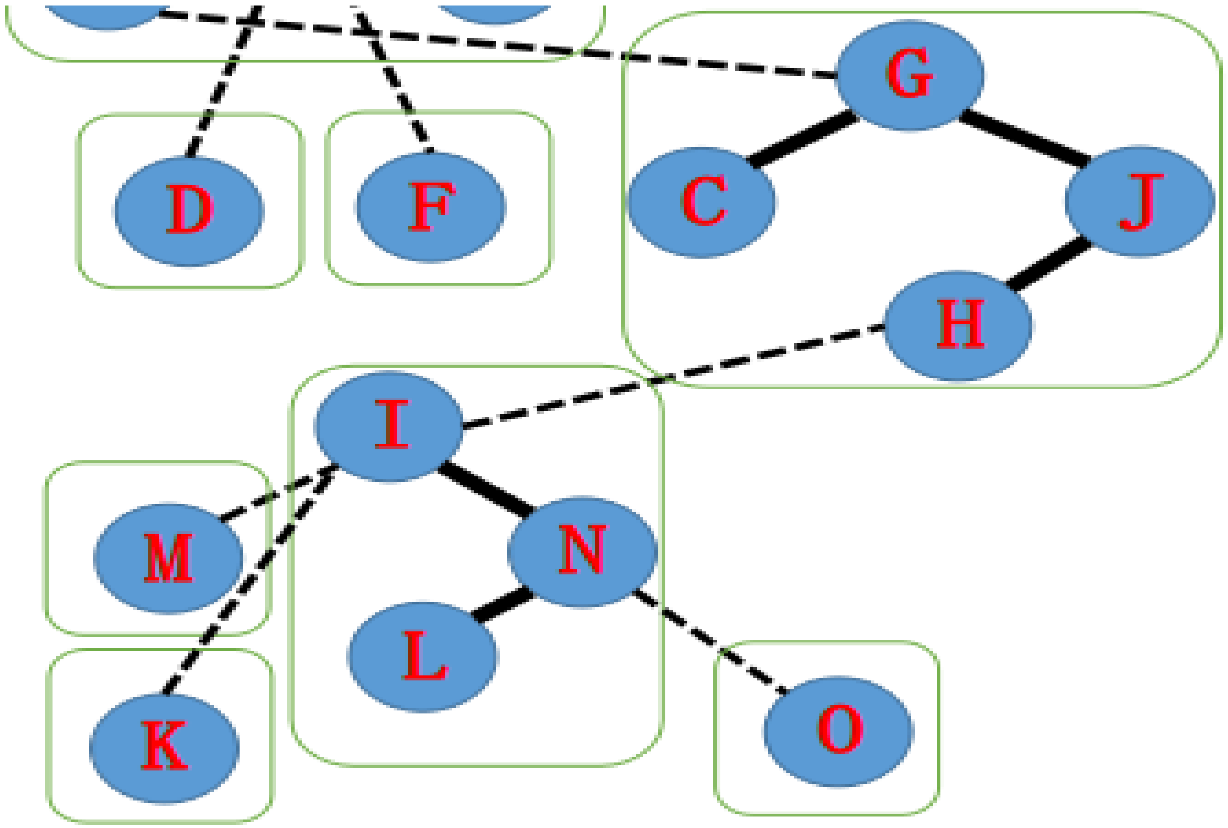
LCT核心操作:access

我们接着把N所属Splay的虚边指向的I（在原树上是L的父亲）也转到它所属Splay的根，splay(I)。

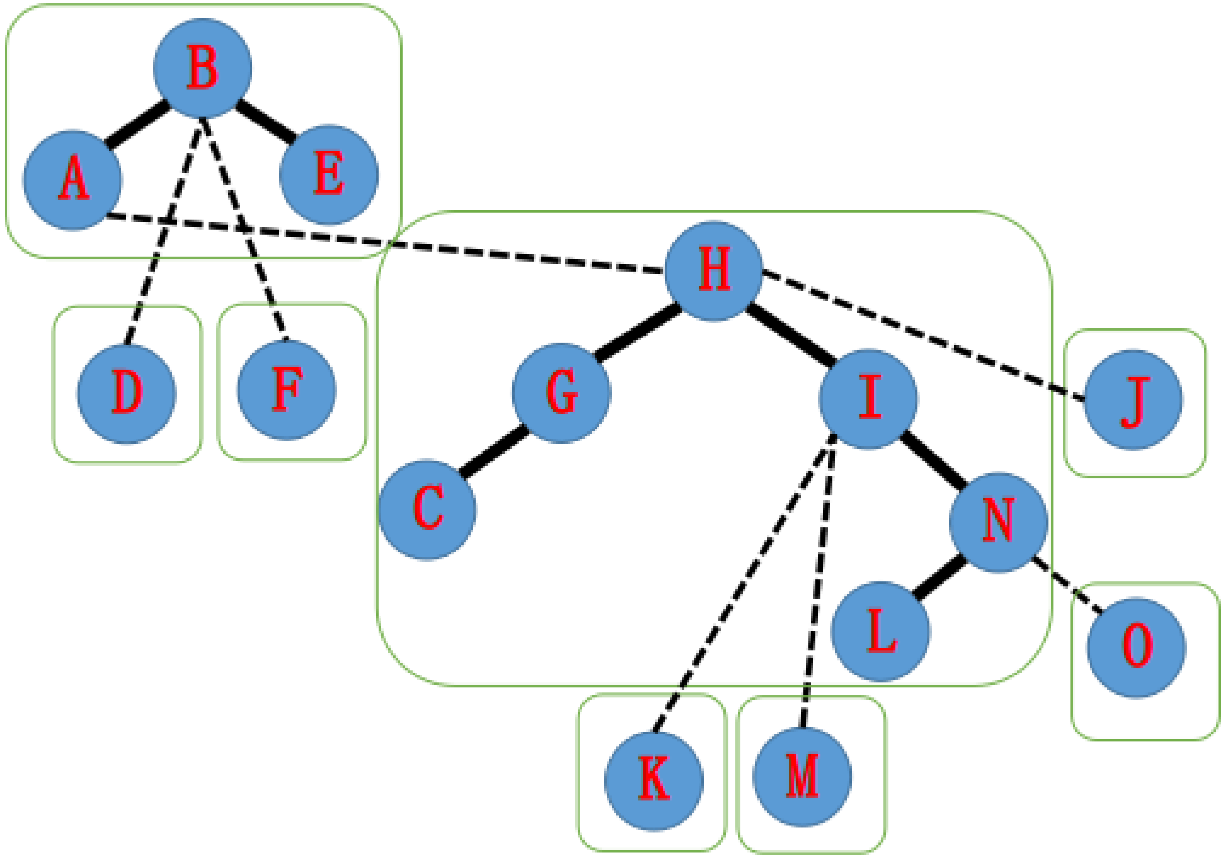
原来在I下方的重边I-K要变轻（同样是将右儿子去掉）。

这时候I-L就可以变重了。因为L肯定是在I下方的（刚才L所属Splay指向了I），所以I的右儿子置为N，满足性质1。

然后就变成了这样——

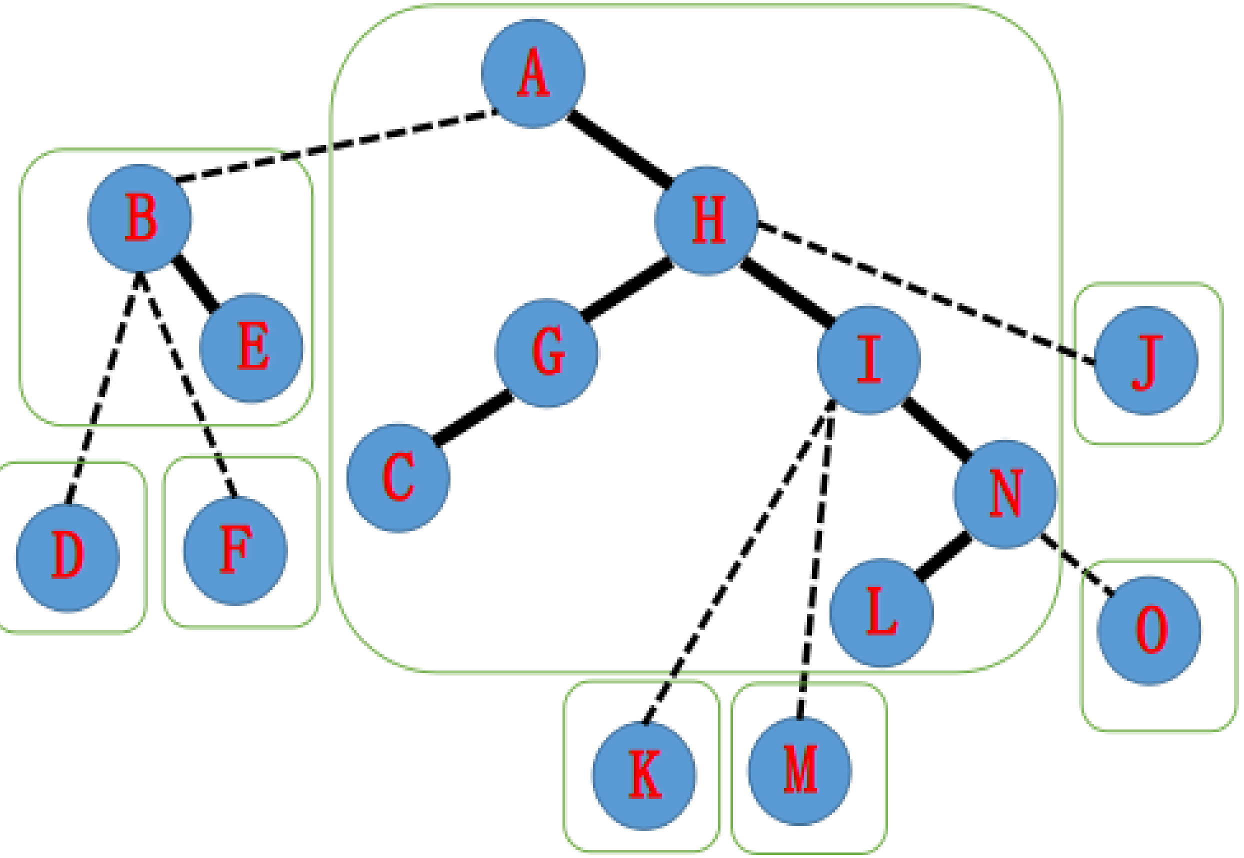


I指向H，接着splay(H)，H的右儿子置为I。



LCT核心操作:access

H 指向A，接着splay(A)，A的右儿子置为H。A-N 的路径已经在一个Splay中了，大功告成！



- 代码其实很简单……循环处理，只有四步——
- 1)转到根；
- 2)换儿子；
- 3)更新信息；
- 4)当前操作点切换为轻边所指的父亲，转1

此处操作以模板题：洛谷 P3690 Link Cut Tree （动态树）为例
题意：维护四种操作
1)询问从x到y的路径上的点的权值的xor和。保证x到y是联通的
2)连接x到y，若x到y已经联通则无需连接
3)删除边(x，y)，不保证边(x，y)存在
4)将点x上的权值变成y

访问access

换根makeroot

只是把根到某个节点的路径拉起来并不能满足我们的需要.

更多时候,我们要获取指定两个节点之间的路径信息。

然而一定会出现路径不能满足按深度严格递增的要求的情况。根据性质1，这样的路径不能在一个Splay中。

access(x) 后x在Splay中一定是深度最大的点。

splay(x)后，x在Splay中将没有右子树（性质1）。于是翻转整个Splay，使得所有点的深度都倒过来了，x没了左子树，反倒成了深度最小的点（根节点），达到了我们换根的目的。

- pushup(int x){ //上传信息
- tag[x]=tag[son[x][0]]^tag[son[x][1]]^a[x];}
- access(int x){
- for(int y=0;x;x=father[y=x])
- splay(x);//只传了一个参数,
- 因为所有操作的目标都是该Splay的根
- son[x][1]=y,pushup(x);}

- pushr(int x) //rev[]为翻转标记
- {swap(son[x][0],son[x][1]);
- rev[x]^=1;}//翻转操作
- makeroot(int x){
- access(x);splay(x);
- pushr(x);}

部分操作：寻根findroot，分裂spilt

findroot:找x所在原树的树根，主要用来判断两点之间的连通性

（findroot(x)==findroot(y)表明x,y在同一棵树中）

- int findroot(int x){
-     access(x); splay(x);
-     while(son[x][0])
-     pushdown(x),x=son[x][0];
- //如要获得正确的原树树根，
- 一定pushdown!
- //同样利用性质1，不停找左儿子，因为其深度一定比当前点深度小。
-     splay(x);//保证复杂度
-     return x;}

split(x,y)定义为拉出x-y的路径成为一个Splay（这里以y作为该Splay的根）

- void split(int x,int y){
-     makeroot(x);
-     access(y);splay(y);}

- //x成为了根，那么x到y的路径就可以用access(y)直接拉出来了，将y转到Splay根后，我们就可以直接通过访问y来获取该路径的有关信息

将x-y的边断开。
如果题目保证断边合法，倒是很方便。
使x为根后，y的父亲一定指向x，深度相差一定是1。
当access(y),splay(y)以后，x一定是y的左儿子，直接双向断开连接
正确姿势——先判一下连通性（注意findroot(y)以后x成了根），再看看x,y是否有父子关系，还要看y是否有左儿子。
因为access(y)以后，假如y与x在同一Splay中而没有直接连边，那么这条路径上就一定会有其它点，
在中序遍历序列中的位置会介于x与y之间。
那么可能y的父亲就不是x了。

连一条x-y的边（本蒟蒻使x的父亲指向y，连一条轻边）

```
bool link(int x,int y){
    makeroot(x);
    if(findroot(y)==x)return 0;
//两点已经在同一子树中,再连边不合法
    father[x]=y; return 1;}
```

- 如果题目保证连边合法，代码就可以更简单

```
void link(int x,int y){
    makeroot(x); father[x]=y;}
```

也可能y的父亲还是x，那么其它的点就在y的左子树中

只有三个条件都满足，才可以断掉。

保证断边合法：

```
void cut(int x,int y){
    split(x,y);
    father[x]=son[y][0]=0;
    pushup(y);//少了个儿子，也要上传一下}
```

不保证断边合法：

```
bool cut(int x,int y){
    makeroot(x);
    if(findroot(y)!=x||father[y]!=x||son[y][0])
        return 0;
    father[y]=son[x][1]=0;//x在findroot(y)后被转到根
    pushup(x); return 1;}
```

补充：如果维护了size，还可以换一种判断

```
bool cut(int x,int y){
    makeroot(x);
    if(findroot(y)!=x||size[x]>2)
        return 0;
    father[y]=son[x][1]=0;
    pushup(x); return 1;}
```

- //解释一下，如果他们有直接连边的话，access(y)以后，为了满足性质1，该Splay只会剩下x,y两个点了。
- 反过来说，如果有其它的点，size不就大于2了么？

//判断节点是否为一个Splay的根:isroot

```
bool isroot(int x){  return
son[father[x]][0]==x ||
son[father[x]][1]==x;}
```

//原理很简单，如果连的是轻边，他的父亲的儿子里没有它