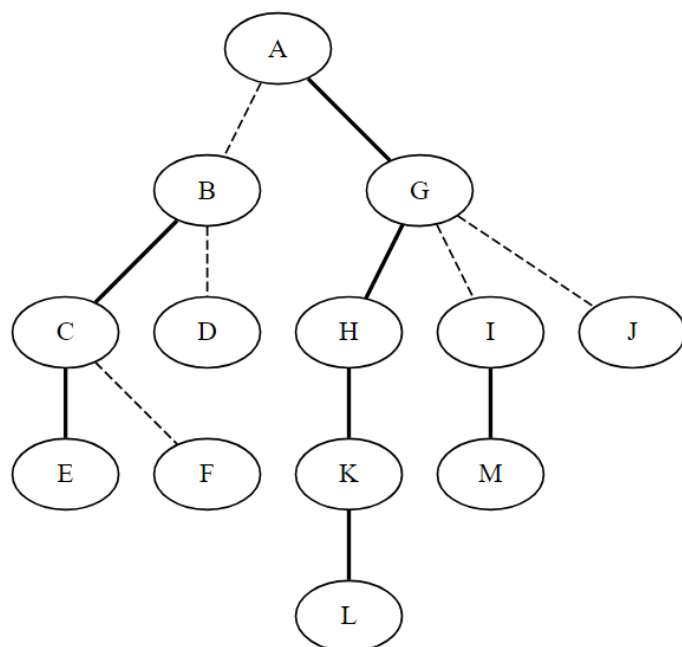


# 问题概述

不知道从什么时候开始很多序列问题都可以“上树”，“上树”的意思就是在树上进行询问。比如把非常常见的区间求和、最值或者带修改区间求和、最值放到树上，变成树上两个点之间的路径上的权值求和、最值。而树链剖分可以在不错的时间内实现序列问题上树的求解，其具体方式就是把树按照某种方式分成多条链，然后把这此链排成一个一维区间，只要我们能够保证不跨链操作，问题就又变回了序列问题。

## 重链剖分

首先，我们定义点  $i$  的父亲是  $f_i$ 、深度是  $dep_i$ 、子树大小为  $size_i$ 。在重链剖分中，我们定义**重边**和**轻边**，对于一个节点  $i$ ，它连向节点个数最大的子树的边是**重边**，其余都是**轻边**。重边连成的链称为重链。比如下图中，实线代表**重边**，虚线代表**轻边**。



可以证明，从一个点到根，最多经过  $O(\log n)$  条轻边。简单证明：首先，从父亲往儿子走，假设从  $x$  走到了  $y$ 。如果经过了一条轻边，那么一定有  $size_y \leq \frac{1}{2} \times size_x$ 。否则这条边一定是重边。那么，我们最多经过  $O(\log n)$  条轻边就会到达叶子。又因为对于一个点到根的路径，轻链和重链一定是交替存在，所以**重链**的数量一定不会超过**轻边**的数量，所以重链的数量也是  $O(\log n)$  的，我们只需要记录每个重链上深度最小的点  $top_i$  和这条重链起末点的  $dfs$  序，然后把这此重链一次放到一个序列里，就可以构成一个在一维序列上查询修改的问题，一般用线段树来维护这个序列。

现在考虑给出树上两个点，如何找到它们之间的路径，首先，如果两个点在同一个重链上，那么我们直接在这条链上操作即可，否则，我们把**链顶深度大**的链上的元素处理完后，从链顶跳到链顶的父亲，从一个重链跳到一个链顶深度更小的重链并重复这一过程，显然我们最后一定可以把这两个点都跳到它们的LCA所在的重链上，这时我们把剩下的元素进行操作，就完成了对两个点之间路径上的元素的操作。我们都知道线段树的操作复杂度是  $O(\log n)$ ，而由我们之前的证明，我们最多操作  $O(\log n)$  个重链，所以树链剖分一次操作的复杂度是  $O(\log^2 n)$ 。

## 例题——洛谷P2590[ZJOI2008]树的统计

### 题目大意

给出一棵有点权的树，要求实现单点修改，查询路径权值和以及查询路径最大权值

### 解题思路

利用树链剖分解决即可，细节可以看代码注释

### 代码实现

```
#include <cstdio>
#include <iostream>
#include <algorithm>
using namespace std;

#define rep(x, y) for(int y = F[x];y;y = Next[y]) //遍历边表的定义

const int maxn = 60100;

int n, X[maxn / 2], deep[maxn / 2], size[maxn / 2], fa[maxn / 2];
int F[maxn], v[maxn], Next[maxn], EID = 1;
int tot = 0, pos[maxn], cc[maxn];
int x, y, ww;
struct Tree{
    int l, r, mx, s;
    Tree()
    {
        mx = -2147483647;
        s = 0;
    }
}node[100005]; //线段树节点定义

inline int read() //读入优化
{
    int a = 0;
    bool flag = false;
    char ch;
    while(!((((ch = getchar()) >= '0') && (ch <= '9')) || (ch == '-')));
    if(ch == '-')
        flag = true;
    else
    {
        a = a * 10;
        a += ch - '0';
    }
}
```

```
while((((ch = getchar()) >= '0') && (ch <= '9')) || (ch == '-'))
{
    a = a * 10;
    a += ch - '0';
}
if(flag)
    a = -a;
return a;
}

inline void add(int f, int t)
{
    Next[EID] = F[f];
    v[EID] = t;
    F[f] = EID++;
}

inline void scanf()
{
    n = read();
    for(int i = 1; i <= n - 1; ++i)
    {
        x = read(), y = read();
        add(x, y);
        add(y, x);
    }
    for(int i = 1; i <= n; ++i)
        X[i] = read();
}

inline void dfs(int x) //处理出一个点所有儿子的大小以及点的深度
{
    size[x] = 1;
    rep(x, i)
    {
        if(v[i] == fa[x])
            continue;
        deep[v[i]] = deep[x] + 1;
        fa[v[i]] = x;
        dfs(v[i]);
        size[x] += size[v[i]];
    }
}

inline void dfsc(int x, int chain) //开始划分重链
{
    int k = 0; ++tot;
    pos[x] = tot;
    cc[x] = chain;
    rep(x, i)
        if(deep[v[i]] > deep[x] && size[v[i]] > size[k]) //找到一个大小最大的儿子
```

```
        k = v[i];
    if(k == 0)
        return;
    dfsc(k, chain); //优先把最大的儿子划分到重链里

    rep(x, i)
        if(deep[v[i]] > deep[x] && k != v[i]) //给剩余的儿子新划分一条重链
            dfsc(v[i], v[i]);
}

inline void build(int k, int l, int r)
{
    node[k].l = l, node[k].r = r;
    if(l == r)
        return;
    int mid = (l + r) >> 1;
    build(k << 1, l, mid);
    build(k << 1 | 1, mid + 1, r);
}

inline void change(int k, int x, int y)
{
    int l = node[k].l, r = node[k].r, mid = (l + r) >> 1;
    if(l == r)
    {
        node[k].s = node[k].mx = y;
        return;
    }
    if(x <= mid)
        change(k << 1, x, y);
    else
        change(k << 1 | 1, x, y);

    node[k].s = node[k << 1].s + node[k << 1 | 1].s;
    node[k].mx = max(node[k << 1].mx, node[k << 1 | 1].mx);
}

inline int qrmx(int k, int x, int y)
{
    int l = node[k].l, r = node[k].r, mid = (l + r) >> 1;
    if(l == x && y == r)
        return node[k].mx;
    if(y <= mid)
        return qrmx(k << 1, x, y);
    else if(x > mid)
        return qrmx(k << 1 | 1, x, y);
    else
        return max(qrmx(k << 1, x, mid), qrmx(k << 1 | 1, mid + 1, y));
}
```

```
inline int qrs(int k, int x, int y)
{
    int l = node[k].l, r = node[k].r, mid = (l + r) >> 1;
    if(l == x && y == r)
        return node[k].s;
    if(y <= mid)
        return qrs(k << 1, x, y);
    else if(x > mid)
        return qrs(k << 1 | 1, x, y);
    else
        return qrs(k << 1, x, mid) + qrs(k << 1 | 1, mid + 1, y);
}
```

```
inline int ss(int x, int y)//查询路径权值和
{
    int sum = +0;
    while(cc[x] != cc[y])
    {
        if(deep[cc[x]] < deep[cc[y]])
            swap(x,y);
        sum += qrs(1, pos[cc[x]], pos[x]);
        x=fa[cc[x]];
    }
    if(pos[x] > pos[y])
        swap(x,y);
    sum += qrs(1, pos[x], pos[y]);
    return sum;
}
```

```
inline int smx(int x, int y)//查询路径最大值
{
    int ret = -2147483647;
    while(cc[x] != cc[y])
    {
        if(deep[cc[x]] < deep[cc[y]])
            swap(x,y);
        ret = max(ret, qrmx(1, pos[cc[x]], pos[x]));
        x=fa[cc[x]];
    }
    if(pos[x] > pos[y])
        swap(x,y);
    ret = max(ret, qrmx(1, pos[x], pos[y]));
    return ret;
}
```

```
inline void solve()
{
    build(1, 1, n);
    for(int i = 1; i <= n; ++i)
        change(1, pos[i], X[i]);
    int q, a, b;
```

```
char opt[20];
q = read();
for(int i = 1; i <= q; ++i)
{
    scanf("%s%d%d", opt, &a, &b);
    if(opt[0] == 'C')
        change(1, pos[a], b);
    else
    {
        if(opt[1] == 'M')
            printf("%d\n", smx(a, b));
        else
            printf("%d\n", ss(a, b));
    }
}

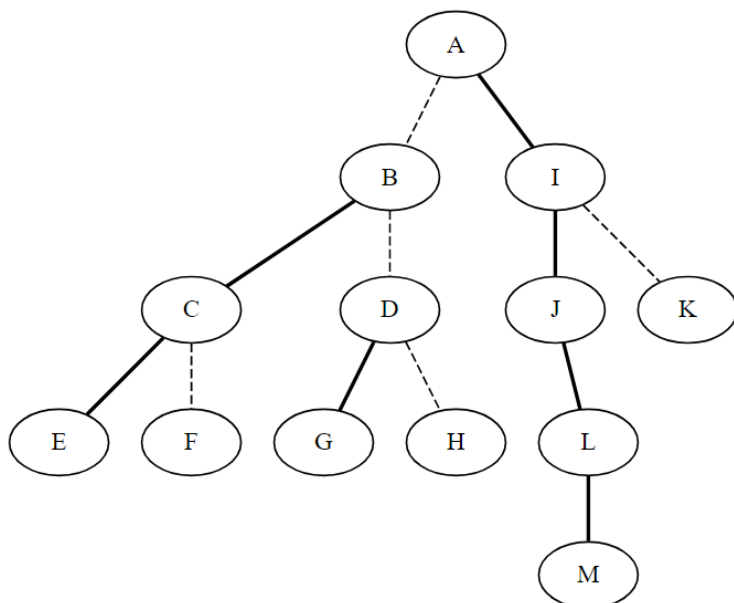
int main()
{
    scanf();
    dfs(1);
    dfsc(1, 1);
    solve();
    return 0;
}
```

## 树链剖分求LCA

在上面我们已经提到，按照跳链的策略，能够把两个点都跳到它们LCA所在的重链上，那么当这两个点都跳到同一个重链上时，我们只需要找其中深度更小的那个就能找到两个点的LCA了，时间复杂度 $O(\log n)$

## 长链剖分

与树链剖分不同，我们还可以按照子树中深度最大的叶子深度最大的点进行长链剖分，例如下图



我们仿照重链剖分的方式分析，假设我们现在从 $i$ 走到了 $f_i$ 并且经过了一条轻边，那么显然 $f_i$ 至少还有一个儿子，所以根节点至少还有一个深度不小于 $dep_i$ 的子树，那么通过计算我们会发现从一个点到根最多经过 $O(\sqrt{n})$ 条轻边和长链。所以，一般情况下，长链剖分的复杂度是不如重链剖分的。但是特定情况下，长链剖分会比重链剖分更优，比如这个问题——在线查询某个点的第 $k$ 个祖先。

首先，一个点的第 $k$ 个祖先所在的长链的长度一定不小于 $k$ ，因为如果小于，那么显然我们把那个祖先所在的长链改成连到这个点的链，一定比之前的长链剖分要长，说明之前的剖分是错误的。有了这个结论，我们先长链剖分并求出倍增数组，然后暴力求出每条长链的链顶的第 $k$ 个祖先以及它向下的 $k$ 个点，由于长链长度和一定是 $n$ ，所以复杂度为 $O(n)$ 。接下来考虑如何在线求出一个节点 $x$ 的第 $k$ 个祖先。首先假设不大于 $k$ 的最大二进制数为 $k_b$ ，则先通过倍增LCA数组从 $x$ 跳至第 $k_b$ 个祖先 $x'$ 处。由于 $k - k_b < \frac{k}{2}$ ，根据我们之前证明的结论， $x'$ 所在的长链长度应不小于 $k_b$ ，也就严格大于 $k - k_b$ ，而我们已经预处理出了 $x'$ 所在长链的顶端向上或向下的不少于 $k - k_b$ 个节点，这一定覆盖了 $x'$ 的第 $k - k_b$ 倍祖先，所以直接利用这个额外信息向上跳即可。

From:

<https://wiki.cvbbacm.com/> - CVBB ACM Team

Permanent link:

<https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:hotpot:treechain>



Last update: 2020/07/22 13:52