虚树 Virtual Tree

- ## 概念[1]

  对于一棵树，如果要对树上的某一个点的集合进行处理，这些要进行处理的点被称为**关键点**。将这些关键点在原来树的边基础上，以**最小代价**[2]连接成一个新的树，这棵树便是虚树。要维护最小代价，很容易想到将所有点的**LCA**加入虚树中，即可以最小代价将这些关键点连接起来。

> [1] 虚树，是对于一棵给定的节点数为n 的树 T，构造一棵新的树 T' 使得总结点数**最小**且**包含指定的某几个节点和他们的LCA**。 ----------【洛谷日报#185】 浅谈虚树
>
> [2] 最小代价指除了关键点外，加入树中的点要尽可能少，也就是构造出的虚树中的结点要尽可能少。

- ## 前置知识

  - DFS序：在一棵树上从某一个节点开始进行**深度优先遍历**，按访问顺序对每个节点进行编号，所得的编号就是DFS序
  - LCA： 最近公共祖先，即在一棵**有根树**上，对于两个结点u和v，距离他们**最近**（距离根**最远**/深度最大）的公共祖先即LCA。对于LCA的求解可以使用**倍增法**实现。

LCA代码：

```cpp
#include<bits/stdc++.h>
using namespace std;
const int maxn = 10005;
int fa[maxn][20], dep[maxn];//fa[i][j]为i节点第2^j的祖先，dep[i]为i节点的深度
int n, root = -1;
vector<int> vec[maxn];//存储原始图

void input()
{
    cin >> n;
    int u, v;
    for (int i = 1; i < n; ++i)
    {
        scanf("%d%d", &u, &v);
        vec[u].push_back(v);
        vec[v].push_back(u);
    }
    root = 1;
    dep[root] = 1;
}

void getdep(int x, int f)
{
    for (auto it = vec[x].begin(); it != vec[x].end(); it++)
```

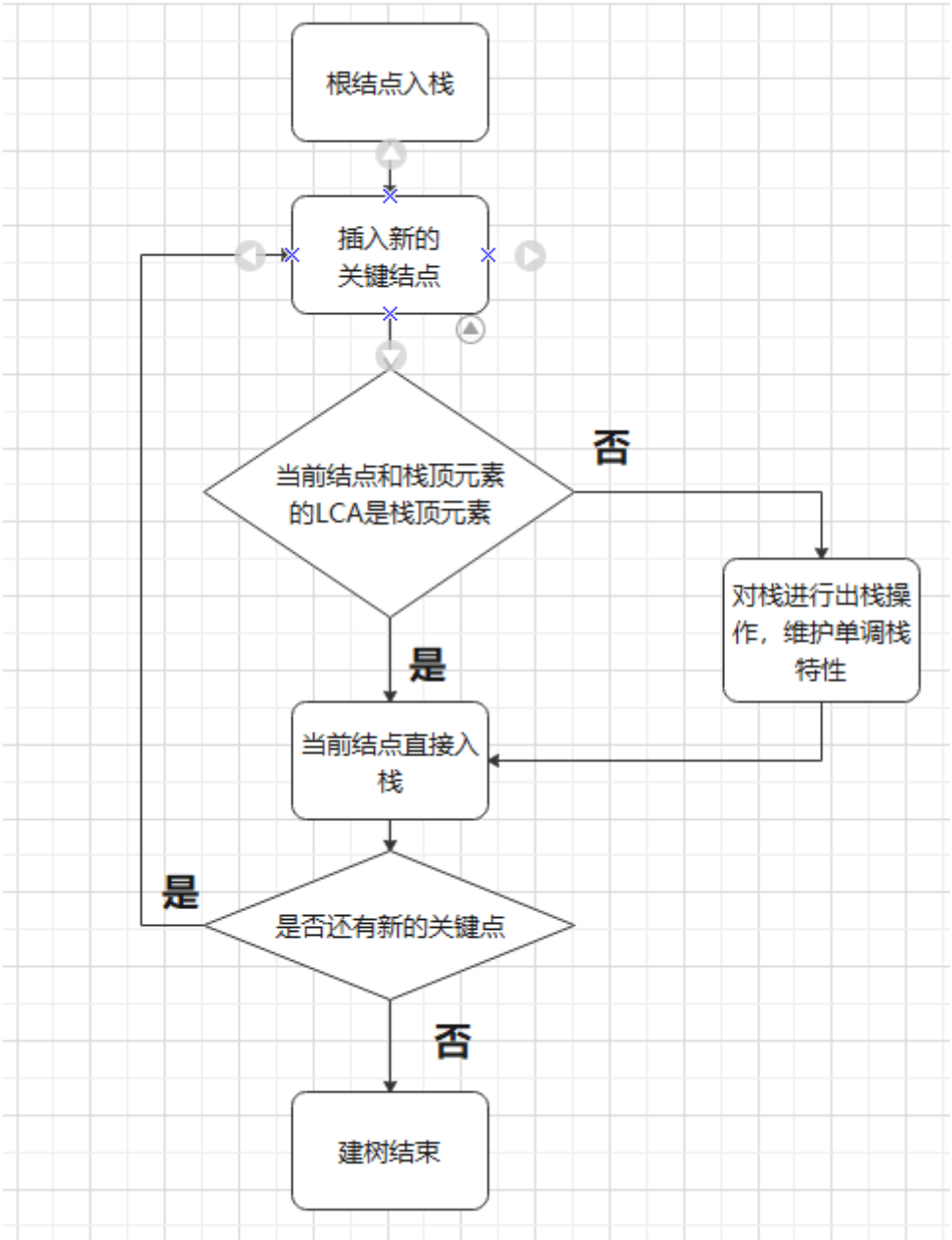
```
    {
        if (*it == f)
        {
            it = vec[x].erase(it);
        }
        else
        {
            dep[*it] = dep[x] + 1;
            fa[*it][0] = x;
            getdep(*it, x);
            it++;
        }
    }
}

void getfa()
{
    for (int i = 1; (1 << i) <= n; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            fa[j][i] = fa[fa[j][i - 1]][i - 1];
        }
    }
}

int LCA(int u, int v)
{
    if (dep[u] < dep[v])
        swap(u, v);
    int i = -1, j;
    while ((1 << (i + 1)) <= dep[u])
        i++;
    for (j = i; j >= 0; --j)
    {
        if (dep[u] - (1 << j) >= dep[v])
        {
            u = fa[u][j];
        }
    }
    if (u == v) return u;
    for (j = i; j >= 0; --j)
    {
        if (fa[u][j] != fa[v][j])
        {
            u = fa[u][j];
            v = fa[v][j];
        }
    }
    return fa[u][0];
}

```

- **关于虚树的建立**

在建立虚树的时候，可以利用**单调栈**维护虚树上的一条从根节点出发的链。在建树之前，对所有的点进行的**DFS序排序**，这样就能保证单调栈能够模拟正常dfs的顺序，而不需要遍历所有的点。可以认为单调栈就是正常dfs过程中系统生成的栈，按照dfs顺序进行递归以及回溯，只不过在递归的时候选择性地忽略了一些点，只对所有的**关键点**，以及所有关键点之间的**LCA**进行遍历，可以节省大量的无用递归。

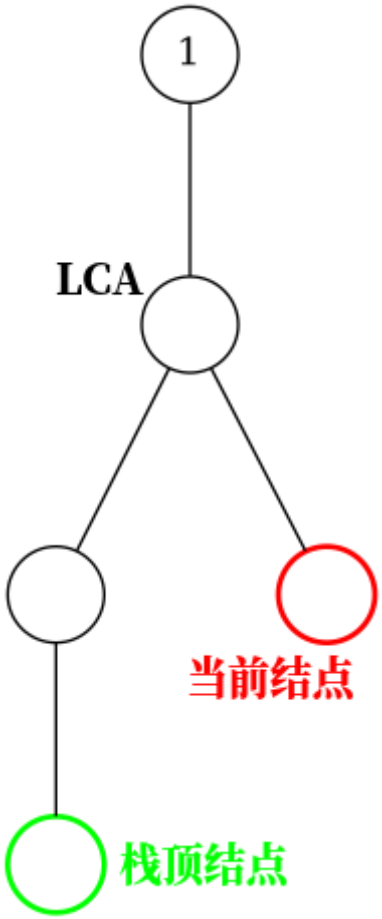
默认虚树和原树具有**相同的根**，这里假设根的编号为***1***，在建树的时候先将根节点强制入栈，也就相当于dfs的递归入口。紧接着按照**DFS序**遍历所有的关键点，由于要维护虚树的正常树形结构，任何两个关键结点的**LCA**必须存在于生成的虚树中，因此在任何一个关键点插入虚树的时候，都需要求得它于上一个关键点的公共祖先位置，并先将公共祖先插入虚树中。

> 建树基本操作流程图



如果当前结点和栈顶元素的LCA是栈顶元素，说明当前结点为栈顶元素的子树结点，即当前结点位于单调栈中维护的虚树的链上，可以直接入栈。

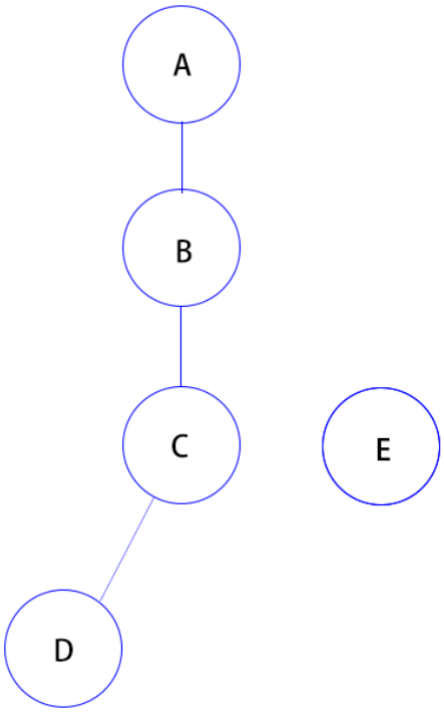
如果当前结点和栈顶元素的LCA并非栈顶元素，那么只会出现如下图一种情况：



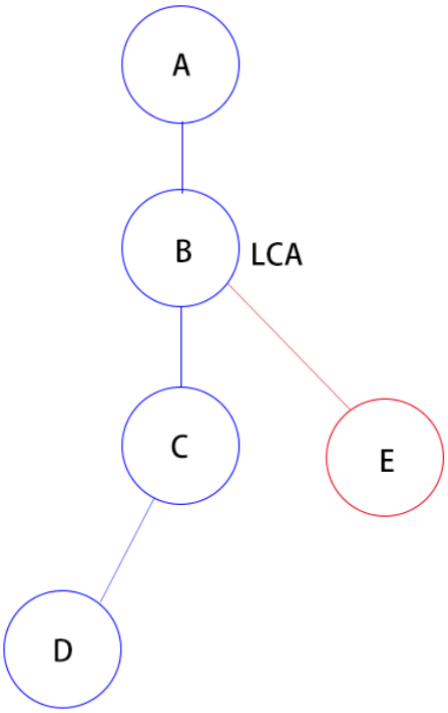
在这种情况下，由于要维护单调栈内存储的是树上从根出发的一条链的性质，那么 LCA之前的所有节点都需要出栈。在出栈的时候，就需要将出栈的结点用图的形式存储 下来，即新生成的虚树的结构。

考虑如下问题，对于图1，ABCD为现在已经春色在于单调栈中的链，而E为要加入的新关键结点，那么对于E与栈顶的LCA，除了E为A的子树结点这种情况之外，存在两种情况：

> 图1

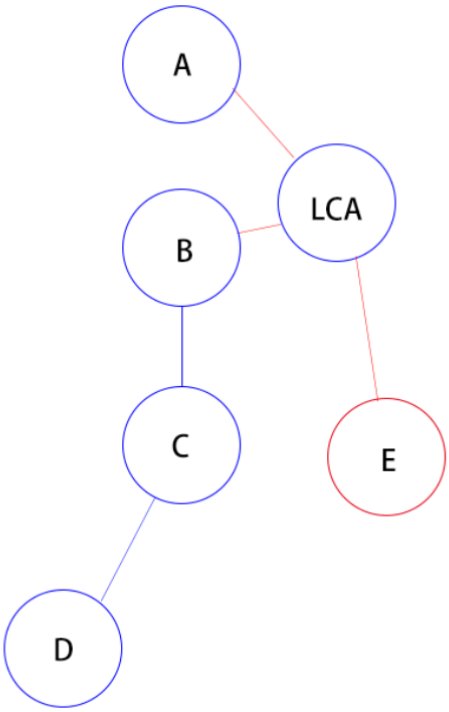


①



第一种情况，当前结点和栈顶结点的LCA已经存在于单调栈中，对于这样的情况，只需要将D和C出栈，紧接着将E入栈即可。在出栈的时候，为了维护虚树的结构，要在CD和BC之间建立有向边，**注意，存储虚树的图和存储原树的图相互分离，互不相交。**最后，将E入栈，结束本次插入。显然，在这种情况下，C和D结点作为LCA结点的子孙节点，其DFS序都大于LCA结点，因此，需要将所有DFS序大于LCA结点的结点出栈。

②



第二种情况就是LCA并未出现在单调栈中，由于单调栈栈底元素为根节点，那么这个未出现在单调栈中的元素必然是根节点的子孙节点，同时它又会是已经在单调栈中的某个元素的祖先节点，同①中的理论，LCA节点的子孙节点的DFS序必然大于它自身，将这些节点全部出栈（**遵循①中的出栈规则**），紧接着，在**LCA结点和最后弹出的结点**之间添加一条边，将LCA节点入栈，最后再将E入栈。

总结如上两种情况，可以得出如下结论：

1. 每次插入新结点的时候，要出栈的结点都是DFS序大于新结点与栈顶结点LCA的结点。
2. 每次插入完成后，栈顶元素都是一个关键结点
3. 插入过程中存在三种情况：

    1>关键结点直接入栈

    2>弹出一部分结点，然后关键结点入栈

    3>弹出一部分结点，然后LCA入栈，最后关键结点入栈

对于插入过程中的三种情况，第一种是最好区分的，条件为**当且仅当新的关键结点与栈顶元素的LCA为栈顶元素**，第二和第三种情况都有一个前置操作就是弹出栈顶DFS序大于LCA结点的元素。在元素弹出这一步骤结束后，才能判断LCA结点是否存在于栈中。考虑出栈时连接结点的情况，正常弹出的时候，将弹出的结点和弹出后的栈顶元素结点连接，但是在**特殊情况下需要将弹出的结点和新插入的LCA结点连接**，为了方便代码的书写，可以手写栈，在弹出结点的时候比较栈顶第二个元素[1]的DFS序以及LCA的DFS序，当栈顶第二元素的DFS序大于LCA的DFS序的时候，栈顶元素必然需要弹出，同时弹出后的栈顶元素同样是要在下一步弹出的，因此可以直接在两个结点之间连边，而不需要考虑插入LCA的情况。

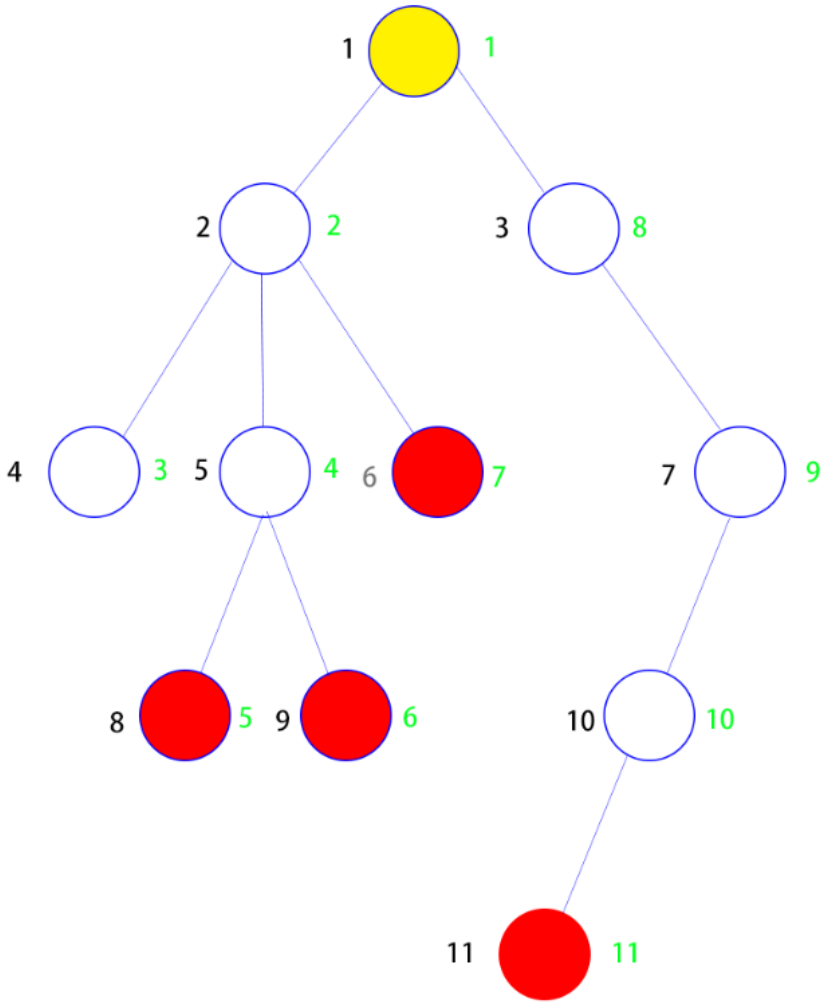
当栈顶第二元素的DFS序不再满足弹出条件的时候，存在两种情况，第一种是栈顶第二元素就是LCA结点，第二种是栈顶第二元素是LCA结点的祖先结点而栈顶元素是LCA的子孙结点，LCA结点需要插入到这两个结点之间。此时，便可以进行特别的判断，如果栈顶第二元素是LCA结点，就在栈顶一二元素之间连边，弹出栈顶元素。否则的话，在LCA和栈顶之间连边，弹出栈顶，插入LCA结点。至此，所有插入的情况分析完毕。

> [1] 由于栈内始终存在一个根节点，同时不会出现任何一个LCA结点的DFS序小于根节点，因此，在插入第一个关键结点之后，永远能够保证单调栈中存在两个或以上的元素。而特殊的，插入第一个关键结点的时候，单调栈内只有根节点一个元素，但是关键结点和根节点的LCA必然是根节点，因此在第一步就可以筛选出来，不会参与到后续的比较栈顶第二元素的过程中。

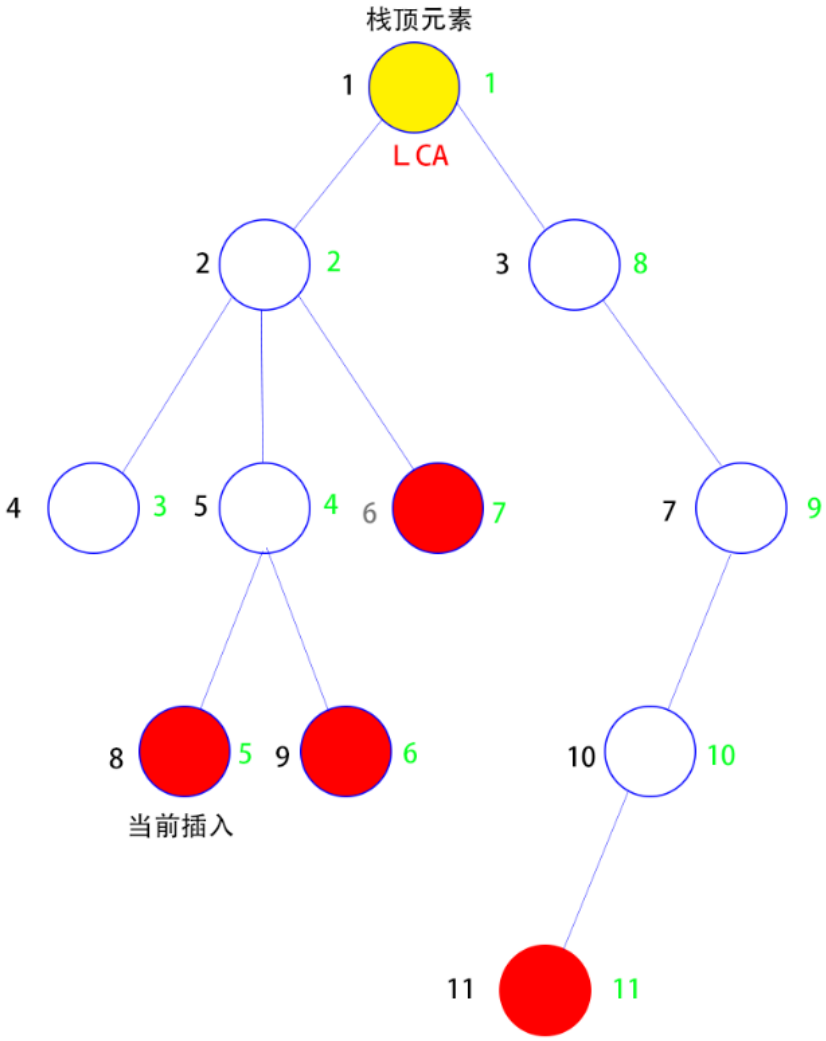
在所有关键结点插入完毕之后，单调栈中剩下了所有从根节点出发，以关键结点为终止的链中DFS序最大的一个，而之前已经出栈的内容已经形成了一棵以原树根节点为根节点的虚树，现在遍历整个栈，将栈中所有元素顺序相连，即可完成虚树的构建。

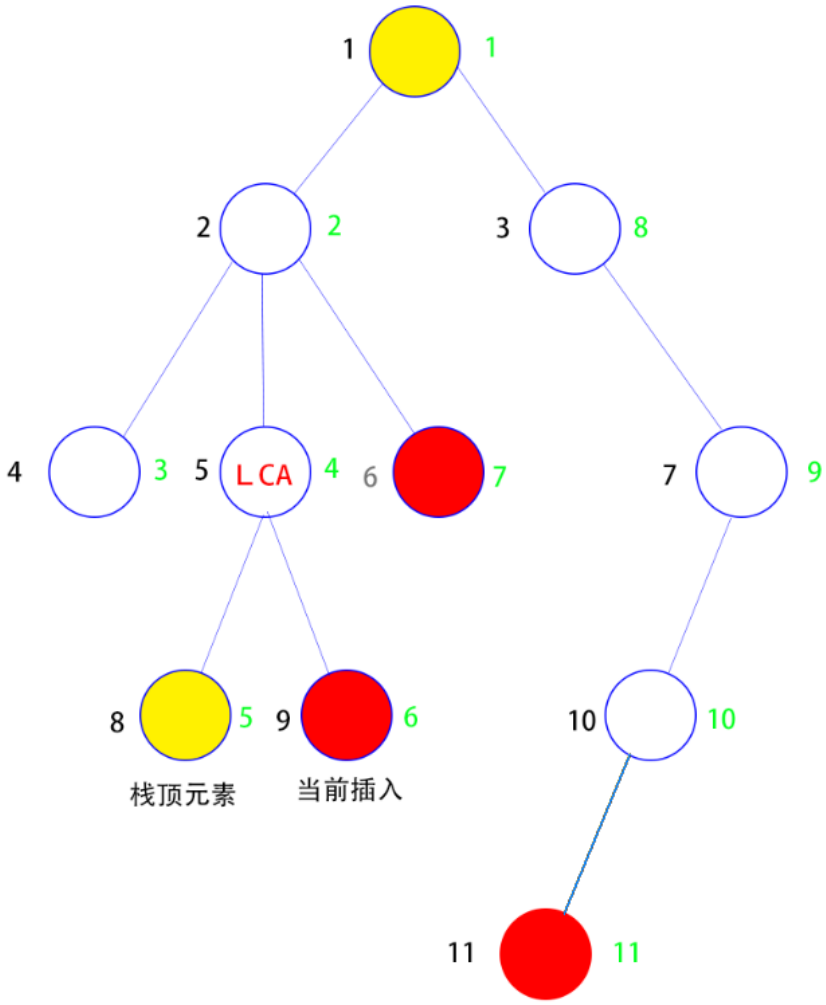
完整流程图如下（暂时没有，被火灾预警搞没了）：

算法模拟：

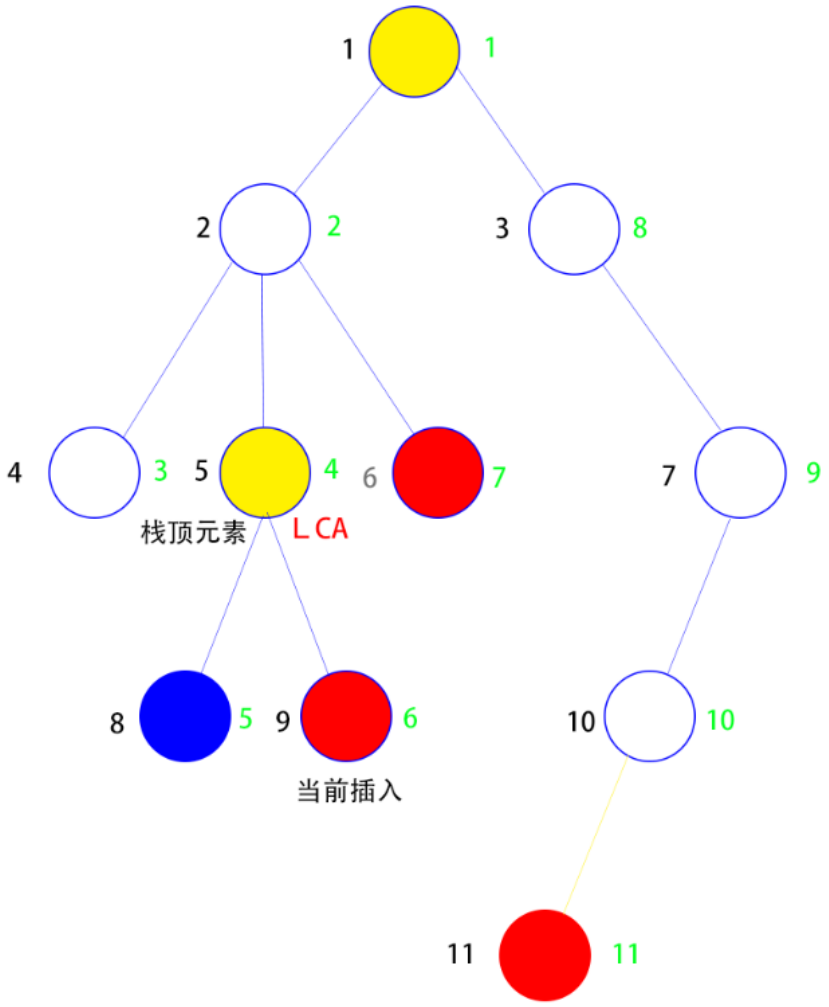


> 黄色为栈内元素，黑色数字为结点编号，绿色为DFS序

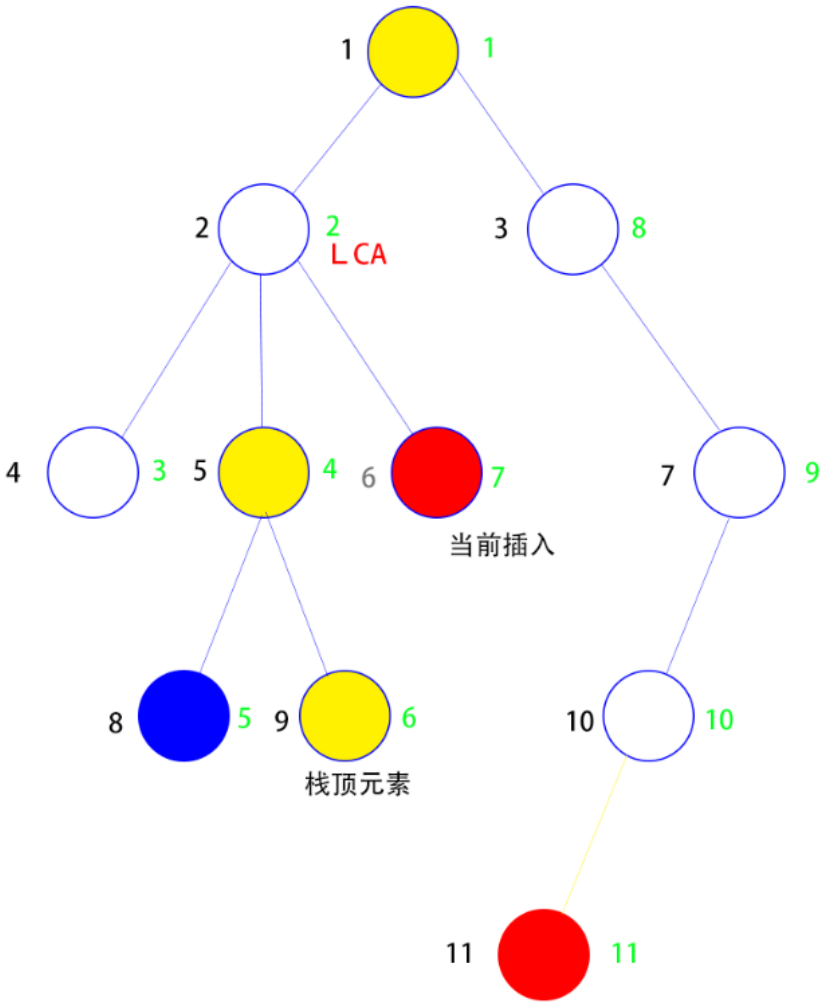

栈顶元素
1
1
LCA
2
2
3
8
4
3
5
4
6
7
7
9
8
5
9
6
10
10
当前插入
11
11



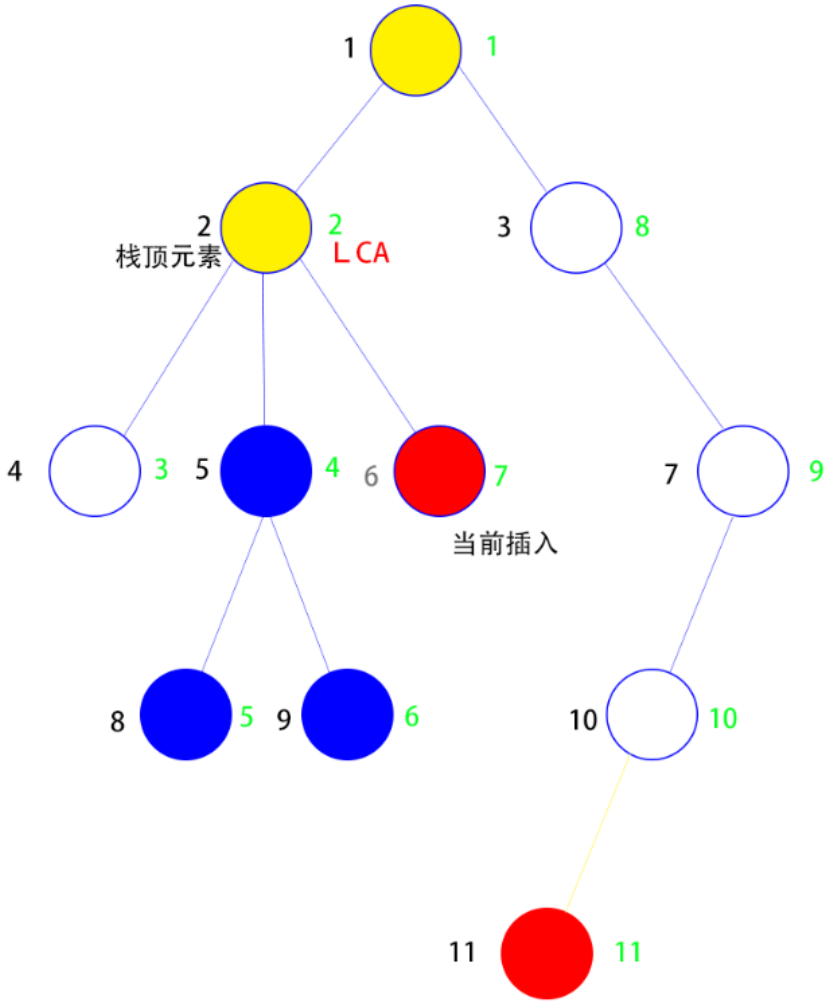
> 栈内元素：1 8



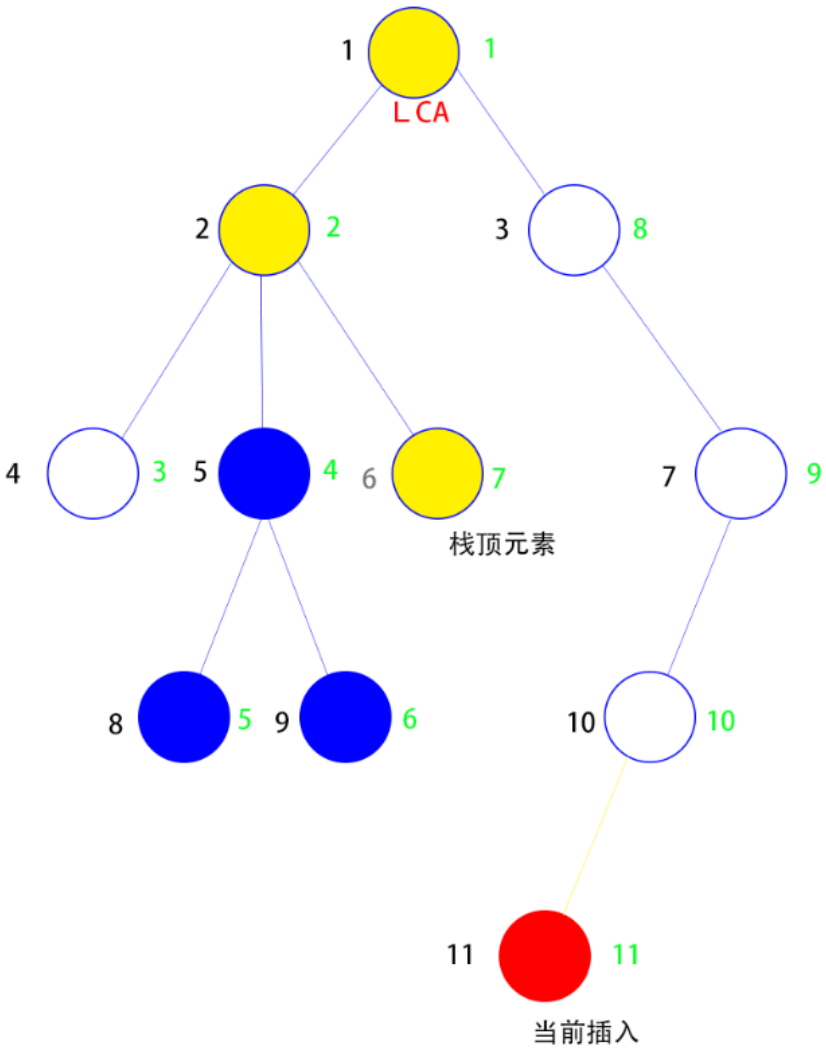
> 8出栈 5入栈



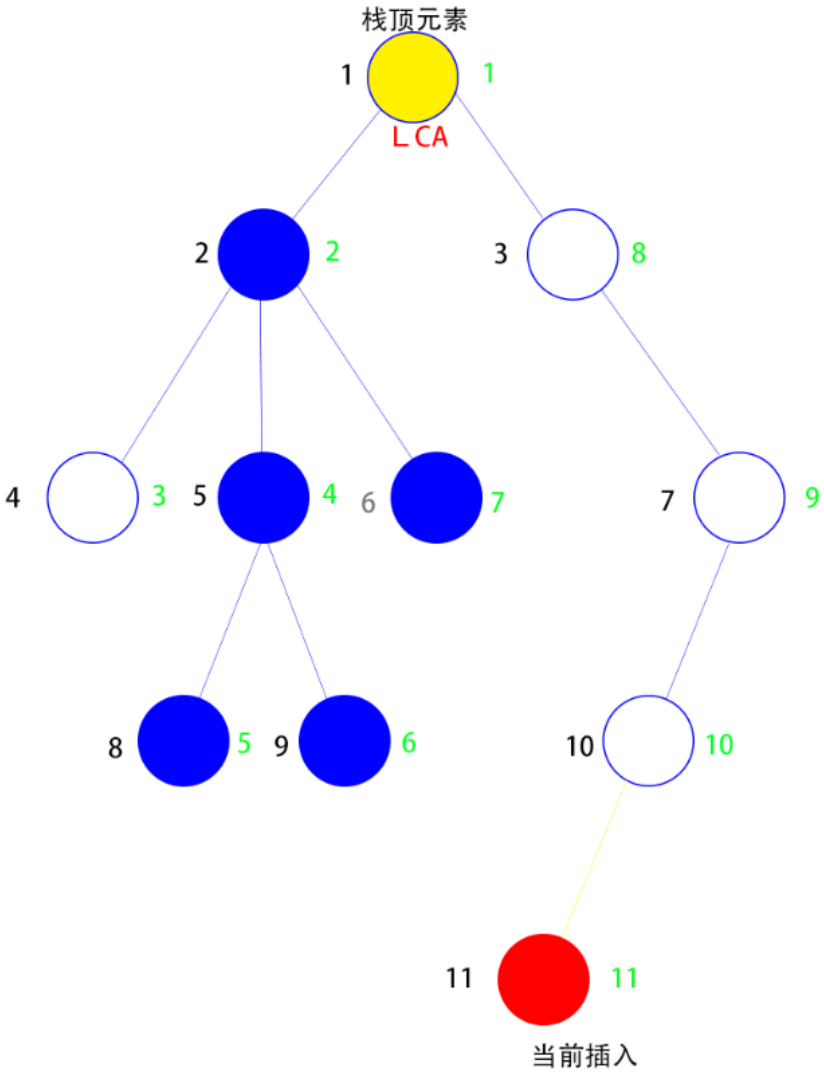
> 9入栈 栈内元素： 1 5 9，已连接的边：5-8



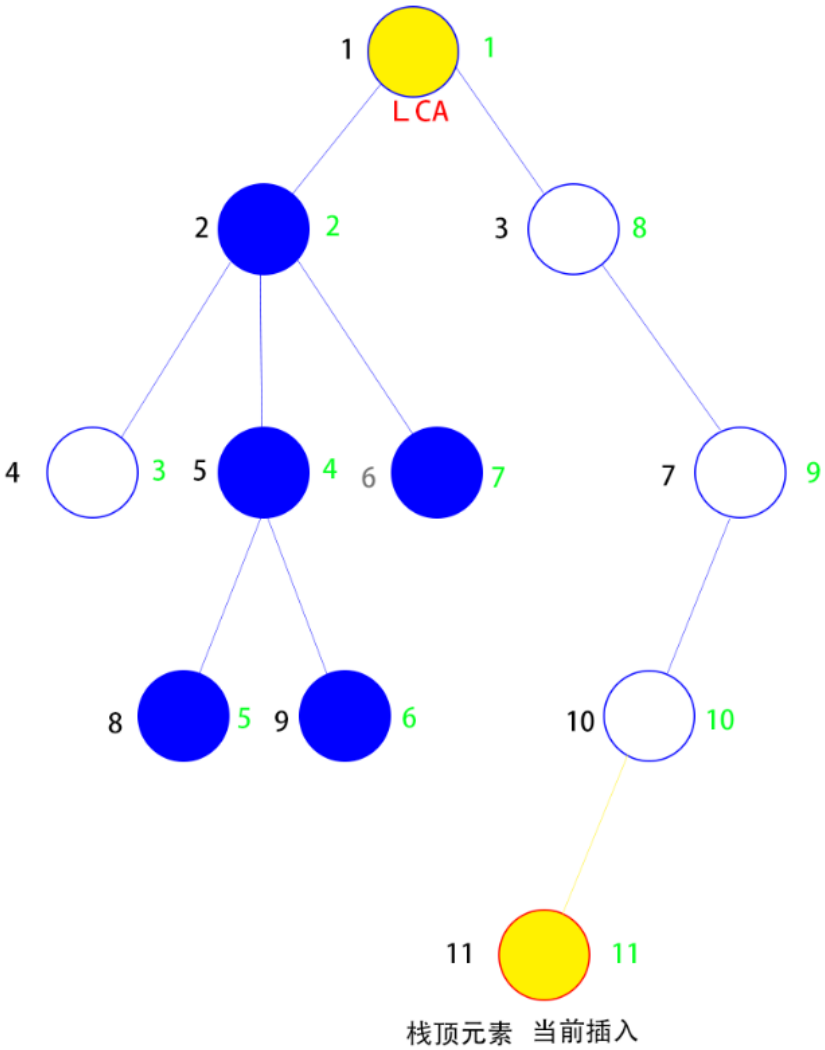
> 9 7 出栈，2入栈 。栈内元素： 1 2 ， 已连接边 5-8，5-9，2-5



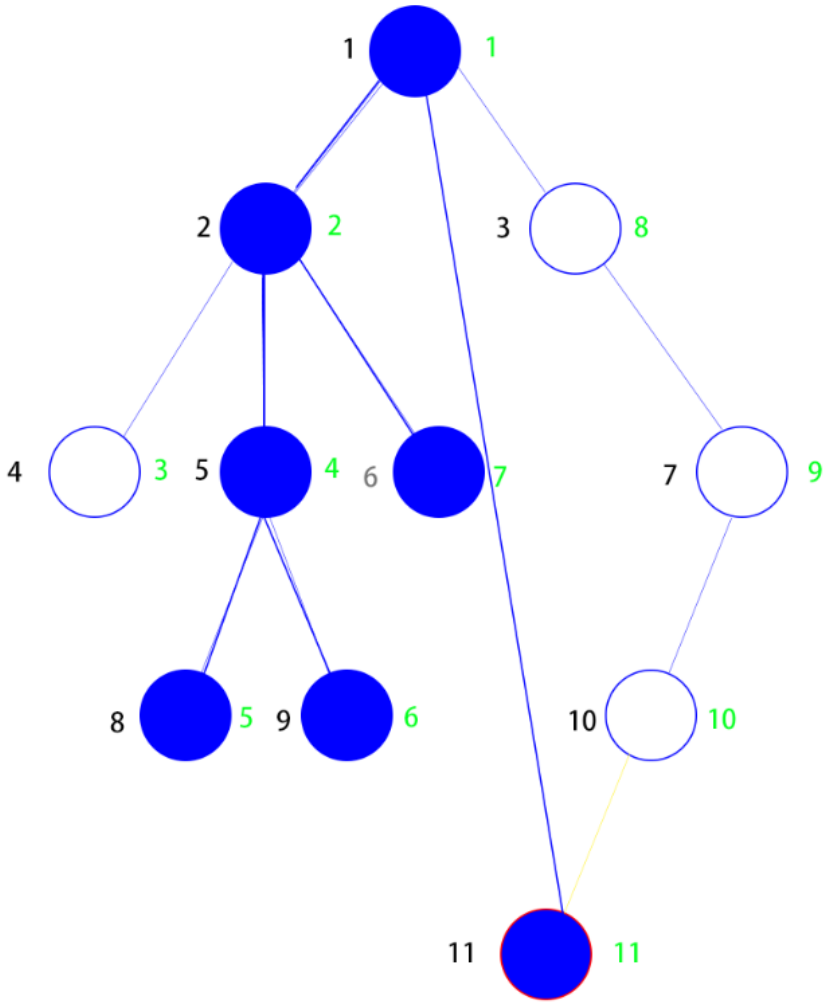
> 结点6入栈



> 6,2结点出栈，栈内元素：1，已连接边：5-9，5-8，2-5，2-6，1-2



> 11号入栈



> 清空栈，连边，虚树建立完毕

- **题目：CF613D Kingdom and its Cities**

  https://www.luogu.com.cn/problem/CF613D

一个王国有n座城市，城市之间由n-1条道路相连，形成一个树结构，国王决定将一些城市设为重要城市。

这个国家有的时候会遭受外敌入侵，重要城市由于加强了防护，一定不会被占领。而非重要城市一旦被占领，这座城市就不能通行。

国王定了若干选择重要城市的计划，他想知道，对于每个计划，外敌至少要占领多少个非重要城市，才会导致重要城市之间两两不连通。如果外敌无论如何都不可能导致这种局面，输出-1

感谢@litble 提供的翻译

输入 #1

```
4
1 3
2 3
4 3
4
2 1 2
3 2 3 4
3 1 2 4
4 1 2 3 4
```

输出 #1

```
1
-1
1
-1
```

输入 #2

```
7
1 2
2 3
3 4
1 5
5 6
5 7
1
4 2 4 6 7
```

输出 #2

```
2
```

对LCA预处理中的dfs进行扩展，额外处理出来DFS序

```
void getdep(int x, int f)
{
    dfn[x] = totdfn++;//
    for (auto it = vec[x].begin(); it != vec[x].end();)
    {
        if (*it == f)
        {
            it = vec[x].erase(it);
        }
        else
        {
            dep[*it] = dep[x] + 1;
            fa[*it][0] = x;
            getdep(*it, x);
            it++;
        }
    }
}
```

建树代码：

```
bool cmp(int l, int r)
{
    return dfn[l] < dfn[r];
}
void build()
{
    for (int i = 0; i < k; i++)
    {
        Main[a[i]] = true;//将所有关键点进行标记
    }
    bool wa = false;//记录是否有解
    for (int i = 0; i < k; i++)
    {
        if (Main[fa[a[i]][0]])//该题目无解的情况仅当关键点中某个结点是另一个结点的父亲结点时
        {
            wa = true;
            break;
        }
    }
    if (wa)
    {
        puts("-1");
    }
    else
    {
        sort(a, a + k, cmp);//按照dfs序排序
        top = 0;//清空栈
        st[++top] = 1;//强制根节点入栈
        head[1] = -1;//初次遇到根结点，进行图的清空，这里图的存储使用了邻接数组
        tot = 0;
        for (int i = 0; i < k; i++)//一共要插入k个关键结点
        {
            if (a[i] != 1)//特判根节点不要重复插入
            {
                int tmp = LCA(a[i], st[top]);//计算lca
                if (tmp != st[top])//判断第一种情况，lca是不是栈顶元素
                {
                    while (dfn[tmp] < dfn[st[top - 1]])//判断栈顶第二元素和lca结点的dfs序，进行弹出操作
                    {
                        add(st[top - 1], st[top]);
                        top--;
                    }
                    if (dfn[tmp] > dfn[st[top - 1]])//判断lca还未出现过的情况
                    {
                        head[tmp] = -1;//清空lca结点对应的邻接数组
                        add(tmp, st[top]);
                        st[top] = tmp;
                    }
                    else//lca已经在栈中的情况
                    {
                        add(tmp, st[top]);
                        top--;
```

```
                    }
                }
                head[a[i]] = -1;//清空当前关键结点邻接数组
                st[++top] = a[i];//入栈
            }
        }
        for (int i = 1; i < top; i++)//清空栈并连接
        {
            add(st[i], st[i + 1]);
        }
        printf("%d\n", dp(1));//递归求解
    }
    for (int i = 0; i < k; i++)Main[a[i]] = false;//重新将所有标记为关键点的点取消标记
}
```

dfs搜索求解：

```
int dp(int x)
{
    int ans = 0;//记录答案
    for (int i = head[x]; i != -1; i = edge[i].ne)
    {
        ans += dp(edge[i].to);//向下递归求和
    }
    if (Main[x])//判断当前点为关键点的情况
    {
        for (int i = head[x]; i != -1; i = edge[i].ne)
        {
            if (Main[edge[i].to])ans++;//如果某个儿子结点是关键点，那么就需要断掉它们之间的一个普通结点，因此ans累加1
            //由于之前已经特别判断过，不会有两个相邻关键结点，因此在原树中，任何两个出现在虚树中的关键结点之间必然存在
            //至少一个普通结点，可以直接删去
        }
    }
    else//当前点不是关键点的情况
    {
        int cnt = 0;//记录儿子中关键点的个数
        for (int i = head[x]; i != -1; i = edge[i].ne)
        {
            if (Main[edge[i].to])cnt++;
        }
        if (cnt == 1)//如果儿子中只有一个关键点，那么就把这个关键点向上迁移到自己身上，显然这样迁移是不会影响答案的
        {
            a[k++] = x;//在迁移的时候，要将自身标记为关键结点，因此把x记录到a数组中，方便随后的清空
            Main[x] = true;
        }
        else if (cnt > 1)ans++;//如果儿子中有多个关键点，就把当前点删除，显然这样是最优且必须的选择
    }
    return ans;//返回答案，此时已经维护了以x为根的子树最优情况
}
```

完整代码：

```cpp
#include<bits/stdc++.h>
using namespace std;
const int maxn = 100005;
int fa[maxn][20], dep[maxn];//fa[i][j]为i节点第2^j的祖先，dep[i]为i节点的深度
int n, root = -1;
int dfn[maxn], totdfn = 1;
bool Main[maxn];
int a[maxn], k;
vector<int> vec[maxn];//存储原始图
struct node
{
    int to, ne;
}edge[maxn];
int head[maxn], tot;
void add(int u, int v)
{
    edge[++tot].to = v;
    edge[tot].ne = head[u];
    head[u] = tot;
}
void input()
{
    cin >> n;
    int u, v;
    for (int i = 1; i < n; ++i)
    {
        scanf("%d%d", &u, &v);
        vec[u].push_back(v);
        vec[v].push_back(u);
    }
    root = 1;
    dep[root] = 1;
}

void getdep(int x, int f)
{
    dfn[x] = totdfn++;
    for (auto it = vec[x].begin(); it != vec[x].end();)
    {
        if (*it == f)
        {
            it = vec[x].erase(it);
        }
        else
        {
            dep[*it] = dep[x] + 1;
            fa[*it][0] = x;
            getdep(*it, x);
            it++;
        }
    }
}

void getfa()
{
    for (int i = 1; (1 << i) <= n; i++)
    {
```

```
        for (int j = 1; j <= n; j++)
        {
            fa[j][i] = fa[fa[j][i - 1]][i - 1];
        }
    }
}

int LCA(int u, int v)
{
    if (dep[u] < dep[v])
        swap(u, v);
    int i = -1, j;
    while ((1 << (i + 1)) <= dep[u])
        i++;
    for (j = i; j >= 0; --j)
    {
        if (dep[u] - (1 << j) >= dep[v])
        {
            u = fa[u][j];
        }
    }
    if (u == v) return u;
    for (j = i; j >= 0; --j)
    {
        if (fa[u][j] != fa[v][j])
        {
            u = fa[u][j];
            v = fa[v][j];
        }
    }
    return fa[u][0];
}


int st[maxn];
int top;
int dp(int x)
{
    int ans = 0;//记录答案
    for (int i = head[x]; i != -1; i = edge[i].ne)
    {
        ans += dp(edge[i].to);//向下递归求和
    }
    if (Main[x])//判断当前点为关键点的情况
    {
        for (int i = head[x]; i != -1; i = edge[i].ne)
        {
            if (Main[edge[i].to])ans++;//如果某个儿子结点是关键点，那么就需要断掉它们之间的一个普通结点，因此ans累加1
            //由于之前已经特别判断过，不会有两个相邻关键结点，因此在原树中，任何两个出现在虚树中的关键结点之间必然存在
            //至少一个普通结点，可以直接删去
        }
    }
    else//当前点不是关键点的情况
    {
        int cnt = 0;//记录儿子中关键点的个数
        for (int i = head[x]; i != -1; i = edge[i].ne)
```

```
        {
            if (Main[edge[i].to])cnt++;
        }
        if (cnt == 1)//如果儿子中只有一个关键点，那么就把这个关键点向上迁移到自己身上，显然这样迁移是不会影响答案的
        {
            a[k++] = x;//在迁移的时候，要将自身标记为关键结点，因此把x记录到a数组中，方便随后的清空
            Main[x] = true;
        }
        else if (cnt > 1)ans++;//如果儿子中有多个关键点，就把当前点删除，显然这样是最优且必须的选择
    }
    return ans;//返回答案，此时已经维护了以x为根的子树最优情况
}
bool cmp(int l, int r)
{
    return dfn[l] < dfn[r];
}
void build()
{
    for (int i = 0; i < k; i++)
    {
        Main[a[i]] = true;//将所有关键点进行标记
    }
    bool wa = false;//记录是否有解
    for (int i = 0; i < k; i++)
    {
        if (Main[fa[a[i]][0]])//该题目无解的情况仅当关键点中某个结点是另一个结点的父亲结点时
        {
            wa = true;
            break;
        }
    }
    if (wa)
    {
        puts("-1");
    }
    else
    {
        sort(a, a + k, cmp);//按照dfs序排序
        top = 0;//清空栈
        st[++top] = 1;//强制根节点入栈
        head[1] = -1;//初次遇到根结点，进行图的清空，这里图的存储使用了邻接数组
        tot = 0;
        for (int i = 0; i < k; i++)//一共要插入k个关键结点
        {
            if (a[i] != 1)//特判根节点不要重复插入
            {
                int tmp = LCA(a[i], st[top]);//计算lca
                if (tmp != st[top])//判断第一种情况，lca是不是栈顶元素
                {
                    while (dfn[tmp] < dfn[st[top - 1]])//判断栈顶第二元素和lca结点的dfs序，进行弹出操作
                    {
                        add(st[top - 1], st[top]);
                        top--;
```

```
                    }
                    if (dfn[tmp] > dfn[st[top - 1]])//判断lca还未出现过的情况
                    {
                        head[tmp] = -1;//清空lca结点对应的邻接数组
                        add(tmp, st[top]);
                        st[top] = tmp;
                    }
                    else//lca已经在栈中的情况
                    {
                        add(tmp, st[top]);
                        top--;
                    }
                }
                head[a[i]] = -1;//清空当前关键结点邻接数组
                st[++top] = a[i];//入栈
            }
        }
        for (int i = 1; i < top; i++)//清空栈并连接
        {
            add(st[i], st[i + 1]);
        }
        printf("%d\n", dp(1));//递归求解
    }
    for (int i = 0; i < k; i++)Main[a[i]] = false;//重新将所有标记为关键点的点取消标记
}
int main()
{
    memset(head, -1, sizeof(head));
    input();
    getdep(1, 0);
    getfa();
    int q;
    cin >> q;
    while (q--)
    {
        scanf("%d", &k);
        for (int i = 0; i < k; i++)scanf("%d", &a[i]);
        build();
    }
    return 0;
}
```

评测记录：

General

#	Author	Problem	Lang	Verdict	Time	Memory	Sent	Judged		
92418870	Practice: i_mailang	613D - 36	GNU C++11	Accepted	171 ms	17592 KB	2020-09-10 13:49:04	2020-09-10 13:49:04	☆	Compare