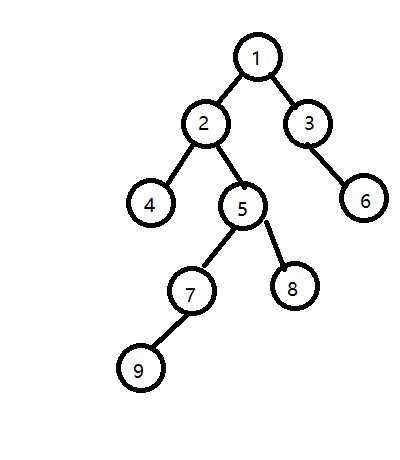
V是树的点的个数，Q是询问次数,h是树平均高度

# 最近公共祖先：

什么是最近公共祖先。最近公共祖先简称LCA（Lowest Common Ancestor），所谓LCA，是当给定一个有根树T时，对于任意两个结点u、v，找到一个离根最远的结点x，使得x同时是u和v的祖先，x 便是u、v的最近公共祖先，因为那个祖先节点离两个节点最近嘛

比如上面这个树，LCA(3,6)=2

## 暴力算法：

暴力法1：向上爬，直到爬到根节点，然后再从根节点起一个个比较，最后一个相同的点是最近的公共祖先

但是一般LCA问题询问会很多，树最坏情况是一条链子，数据是10万，

时间复杂度3\*h\*Q

显然这么做时间太慢

暴力法2：

预处理:对树dfs一遍，每个点记录深度

之后呢，对于每次询问LCA，分为问的两个点深度相同和不同,如果不同，就把深度较大的向上走，到深度相同为止。

然后对于深度相同的两个点，同时向上跳，直到他们2个点相遇为止，相遇的点就是公共祖先。

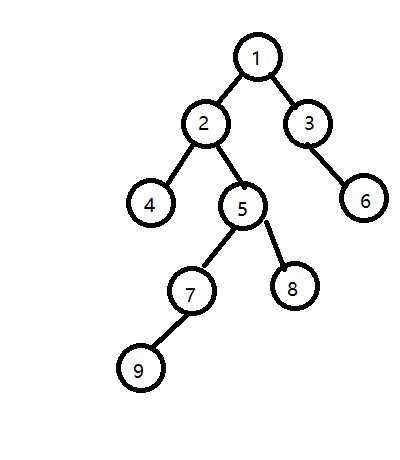
预处理时间复杂度V,总复杂度是V+h\*Q

## tarjan算法：

tarjan算法解决LCA问题是离线算法，必须先知道输入数据。

假设树是如下形式

设我们要查找最近公共祖先的点为9--8，4--6，7--5，5--3；



### 准备：

* 一个邻接表adList，储存图
* 一个并查集，维护各个点关系，他们初始都各自是一个集合

1 2 3 4 5 6 7 8 9

* 一个邻接表li，储存的是询问时候，哪个点能到达哪个点，本题以9--8，

4--6，7--5，5—3建立邻接表

* vis数组,vis[u]表示u有木有访问过，初始都是0，是求公共祖先用的lcavis

### 步骤：

从根节点做dfs，能走就走，在走完一层的时候dfs出来执行以下操作

* 更新并查集中当前节点和子节点的关系，令它们属于同一集合
* 在li中查找与当前节点u有关系的节点v，如果找到，说明当前节点u和找到的节点v是一对要询问的点，再看看v是否已经被访问了，没有就什么都不做，一旦v已经被访问了，说明dfs时已经走过它了，那么它在并查集已经被更新过了，当前节点u和那个节点v的公共祖先就是并查集的根。正确性可以想一下dfs的性质和顺序。这里主意并查集一定能用启发式合并，必须用朴素的合并法，否则打乱了根节点。导致结果不对

**1从根节点开始**

**2遍历该点u所有子节点v，并标记这些子节点v已被访问过。**

**3若是v还有子节点，返回2，否则下一步。**

**4合并v到u上。**

**5寻找与当前点u有询问关系的点v。**

**6若是v已经被访问过了，则可以确认u和v的最近公共祖先为v被合并到的父亲节点a。**

### 模拟过程：

取1为**根节点**，**往下搜索**发现有两个儿子2和3；

先搜2，发现2有两个儿子4和5，

先索4，发现4**没有子节点**，则寻找与其有关系的点；邻接表查询发现6与4有关系，但是**vis[6]=0**，即6还没被搜过，所以**不操作**；发现没有和4有询问关系的点了，返回此前一次搜索，**更新vis[4]=1**；表示4已经被搜完， 合并4和它的父节点2

回溯到2，之后继续**搜5**，发现5有两个儿子7和8;

先**搜7**，发现7有一个子节点9，

**搜9**，发现没有子节点，寻找与其有关系的点，发现8和9有关系，但是**vis[8]=0**,不操作，查询邻接表，发现没有和9有询问关系的点了，返回此前一次搜索，**更新vis[9]=1**；表示9已经被搜完,合并9和它的父节点7，

回到7，发现7没有没被搜过的子节点了，寻找与其有关系的点，查询邻接表，发现5和7有关系，但是**vis[5]=0**，所以**不操作**，**更新vis[7]=1**；

回到5，搜索8，发现8没有子节点，查询邻接表，发现9与8有关系，**此时vis[9]=1**，那么要更新8和9最近公共祖先了，是调用并查集的getfather(9)方法，LCA(8,9)=5，因为9的根节点5。发现没有与8有关系的点了，返回此前一次搜索，**更新vis[8]=1**。

回到5回到，查询邻接表发现7和5有关系，且**此时vis[7]=1**,那么要更新7和5最近公共祖先了，是调用并查集的getfather(7)方法，LCA(7,5)=5，因为7的根节点5。5还和4有关系，但是vis[3]=0,之后没有与5有关系的点了，返回此前一次搜索，**更新vis[5]=1**，合并5和2

回到2，2子节点都被搜完，寻找与其有关系的点，又发现没有和2有关

系的点，**更新vis[2]=1，**合并2和1

回到1，发现3没搜，搜3，然后搜6.。

邻接表里6有关的点是4，**此时vis[4]=1**，那么要更新4和6最近公共祖先了，是调用并查集的getfather(4)方法，LCA(6,4)=1，因为4的根节点是1，发现没有与6有关系的点了，返回此前一次搜索，**更新vis[6]=1，合并6和3**

回到3，发现3没有没被搜过的子节点了，则寻找与3有关系的点，查询邻接表5和3有关系，**此时vis[5]=1，**那么要更新5和3最近公共祖先了，是调用并查集的getfather(5)方法，LCA(5,3)=1，因为5的根节点1。更新vis[3]=1,和并3和1，之后推出dfs，因为搜索完了

### 细节：

* 需要提前知道给的图是不是树，这个算法基于树，
* 给的树如果是无向的，dfs防止向父节点走，应该传个父节点参数，每次递归判断是不是父节点。
* Tanjar算法是离线算法，收集查询统一输出，为了使得输出顺序和查询lca的顺序一致，不用直接输出，而是存在li这个表里，最后再按照输入数据输出，如果担心链表查找效率，可以再开hashmap来索引每一个的下标。
* 时间复杂度是h\*(V +Q), tarjan是各种lca算法最快的，但是局限性是需要离线处理，某些题不适用

## 倍增算法：

上面tarjan是离线算法，需要知道所有询问统一输出，而倍增算法是在线算法，直接对于每次查询输出即可，但是时间比tarjan稍慢。

### 思路：

倍增算法基于**暴力2**的思路,但是每次向上不是走一个点，而是2^i次方个点

预处理：通过dfs遍历，记录深度h[u],同时构造倍增数组p[u][i],表示u的向上走2^i步，是哪个点。P[u][maxLen]表示最大步数，要刚好超过树的最大深度

对于每次询问，还是分为两个点深度相同和不同。假设深度不同，我们要变成深度相同。

变成相同深度：对于较深的点a和较浅的点b,a从2^maxLen不是开始试验，如果走完深度大于了b，就试验2^（maxLen-1）的步数，以此类推。如果刚好等于b的深度，当前就说明达到目的，但是出现2^（i-1）深度小于b的，2^i又大于b的，就把a走2^（i-1）的步数继续试验。直到成功

找lca: 2个点同时向上走2^（maxLen-1）的步数，看看是否超出范围或者是相同节点？是的话，试验2^（maxLen-1）步数；不是的话，出现，a和b同时走2^i是同一节点，而2^(i-1)不是相同节点，就2个点都走2^(i-1)步数，继续上面的走法去实验，直到出现以下情形结束：

2个点向上走一步正好是相同节点。

### 细节分析：

* + 不同题目输入方式不同，树也可以分为有向的和无向，无向树任何节点都能当做根，对于不同根lca结果当然不同，所以题目会给定根节点，dfs时麻烦一些要判断不能往父节点走(往回走)。

有向树需要自己判断根，通过p[u][0]存不存在就可以知道，因为根节点没有任何父亲，p[root][0]不存在

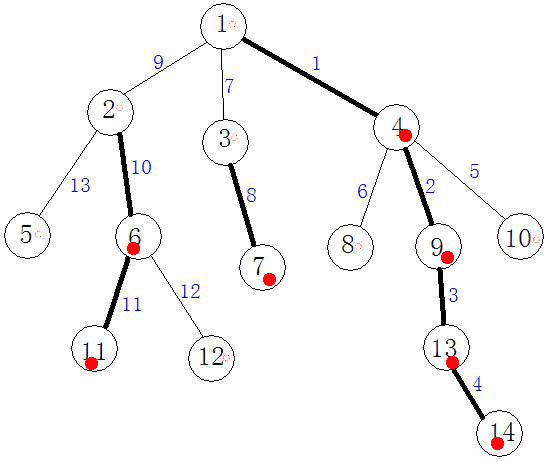
* + 时间复杂度是V+log(h)\*Q,慢于tarjan，和树链剖分时间复杂度相当，空间多于树剖多，在于p[][]这个二维数组，不过一般题目不会卡空间，好处是在线就能用。而且倍增的思想很重要
  + 代码实现分为两种，vector和普通数组，普通数组要单开一个len数组维护count(p[u]) ，使用vector的话不用维护count(p[u])，因为vector<int>p[MAX],后p[u].size();就可以访问长度。效率肯定是数组的强于vector。Vector为了优化效率，可以用vector的reserve函数预定义内存。但仍然不如数组快
  + 优化时间复杂度常数，可选的方法是：在于dfs里，dfs参数尽可能少，不要传深度。用inline函数标记。倍增数组P不用vector,用数组存。领接表用list存，因为有些题不需要频繁访问表中间的元素，list增加删除的效率高于vector

## 树链剖分：

### 思路：

树链剖分的思想类似倍增，每次尽可能走比较多的步数，而不是一步一步往上走。

但是方式和倍增不同。树剖的基本思路转到另一个模板。



对于每次查找:如果两点a和b，在同一条链，公共祖先自然就是深度较小的，不在同一条链子：看a和b的它们的top节点，取top[a]和top[b]深度较大的等于top的父节点。上面图中1 4 9 13 14这条重链最顶是4,4父节点是1，跳到1。

重复此操作直到a和b属于同一链子。此外树链剖分为什么要分轻链和重链，是要让分的链子数尽可能少，这样可以优化更多的时间。

### 细节分析：

* 不同题目输入方式不同，树也可以分为有向的和无向，无向树任何节点都能当做根，对于不同根lca结果当然不同，所以题目会给定根节点，dfs时麻烦一些要判断不能往父节点走(往回走)。有向树需要自己判断根，方法是提前标记p[]数组，通过p[u]是不是0就可以知道，因为根节点没有任何父亲，p[root]是全局数组默认值0
* 时间复杂度是V+log(h)\*Q,慢于tarjan，但也是在线算法，时间和倍增相当。

不过完全是两种思路， 在特殊数据上，如果距离正好是2的倍数，倍增快，如果恰好分成的链子上肯定树剖快。

* 代码实现分为两种，记录son的和记录isheavy的，son[u]代表当前节点u子节点中sz最大的点编号。 Isheavy[u]代表u是不是重节点，也是通过找它的父亲，看父亲中sz最大的是不是自己。

两种方法就是为了区分某个点是不是重节点，最终目的是为了标记top[]数组

经过测试建议用son的方式，时间稍微快于isheavy

* 对于寻找a,b的lca的操作，不要管a和b两个点深度，而是去找他们各自顶节点的深度，即，p[top[a]]和p[top[b]],也不要管他们如果谁不属于任何链子该怎么办，不属于任何链子的点的top必然等于自身。向上走也肯定只会走1个。
* 优化时间复杂度常数，可选的方法是：在于dfs里，dfs参数尽可能少，不要传深度。用inline函数标记。倍增数组P不用vector,用数组存。邻接表用list存或者直接链式前向星，有些题不需要频繁访问表中间的元素，list增加删除的效率高于vector

## RMQ算法：

所谓RMQ算法，就是RMQ (Range Minimum/Maximum Query)：对于长度为n的数组A，回答若干询问RMQ(A,i,j)(i,j<=n-1)，返回数组A中下标在i,j范围内的最小（大）值， RMQ问题是指求区间最值的问题。最简单的方法，就是遍历数组直接搜索，但是这种方式时间复杂度是O(n)。对于数组长度较大，性能要求高的场景不适用。

RMQ的算法有多种比如线段树，树状数组。。。老套路不说了