【POJ1330】最近公共祖先（LCA）：并查集+深搜

最近公共祖先（LCA）问题常见于各种面试题中，针对不同情况算法也不尽相同。

情况1：二叉树是个二叉查找树，且root和两个节点的值(a, b)已知。

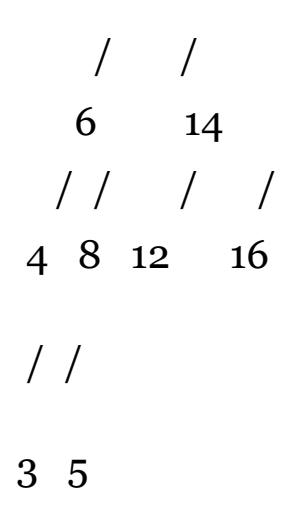
如果该二叉树是二叉查找树，那么求解LCA十分简单。

基本思想为：从树根开始，该节点的值为t，如果t大于t1和t2，说明t1和t2都位于t的左侧，所以它们的共同祖先必定在t的左子树中，从t.left开始搜索；如果t小于t1和t2，说明t1和t2都位于t的右侧，那么从t.right开始搜索；如果t1<=t<= t2，说明t1和t2位于t的两侧（或t=t1，或t=t2），那么该节点t为公共祖先。

情况2：普通二叉树，root未知，但是每个节点都有parent指针。

基本思想：分别从给定的两个节点出发上溯到根节点，形成两条相交的链表，问题转化为求这两个相交链表的第一个交点，即传统方法：求出linkedList A的长度lengthA， linkedList B的长度LengthB。然后让长的那个链表走过abs(lengthA-lengthB)步之后，齐头并进，就能解决了。

情况3：也是最普通的情况，二叉树是普通的二叉树，节点只有left/right，没有parent指针。



基本思想：记录从根找到node1和node2的路径，然后再把它们的路径用类似的情况一来做分析，比如还是node1=3,node2=8这个case.我们肯定可以从根节点开始找到3这个节点，同时记录下路径3,4,6,10，类似的我们也可以找到8,6,10。我们把这样的信息存储到两个vector里面，把长的vector开始的多余节点3扔掉，从相同剩余长度开始比较，4!=8, 6==6,我们找到了我们的答案。

```
1. #include <vector>
2. bool nodePath (bstNode* pRoot, int value, std::vector<bstNode*>& path)
3. {
4.     if (pRoot==NULL) return false;
5.     if (pRoot->data!=value)
6.     {
```

```
7.      if (nodePath(pRoot->pLeft,value,path))
8.      {
9.          path.push_back(pRoot);
10.         return true;
11.     }
12.     else
13.     {
14.         if (nodePath(pRoot->pRight,value,path))
15.         {
16.             path.push_back(pRoot);
17.             return true;
18.         }
19.         else
20.             return false;
21.     }
22. }
23. else
24. {
25.     path.push_back(pRoot);
26.     return true;
27.  }
28. }
29. bstNode* LCAC(bstNode* pNode, int value1, int value2)
30. {
31.     std::vector<bstNode*> path1;
32.     std::vector<bstNode*> path2;
33.     bool find = false;
34.     find |= nodePath(pNode, value1, path1);
35.     find &= nodePath(pNode, value2, path2);
36.     bstNode* pReturn=NULL;
37.     if (find)
38.     {
39.         int minSize = path1.size()>path2.size()?path2.size():path1.size();
40.         int it1 = path1.size()-minSize;
41.         int it2 = path2.size()-minSize;
42.         for (;it1<path1.size(),it2<path2.size();it1++,it2++)
43.         {
44.             if (path1[it1]==path2[it2])
45.             {
```

```
46.             pReturn = path1[it1];
47.             break;
48.         }
49.     }
50.   }
51.   return pReturn;
52. }
```

下面说一下本文的题目，也就是**POJ1330**，用网上流行的**LCA**算法**Tarjan**求解（并查集+深搜）。

LCA是求最近公共祖先问题， tarjan的算法是离线算法，时间复杂度为O(n＋Q)，n为数据规模，Q为询问个数
其中用到并查集。关键是dfs的主循环比较重要。离线算法就是对每个查询，都要求以下，此算法在lrj的黑书中简单提起过，后边还有O（n）- o（1）的算法，正在研究中。。。

分类，使每个结点都落到某个类中，到时候只要执行集合查询，就可以知道结点的LCA了。
对于一个结点u，类别有　以u为根的子树、除类一以外的以f(u)为根的子树、除前两类以外的以f(f(u))为根的子树、除前三类以外的以f(f(f(u)))为根的子树.....
类一的LCA为u,类二为f(u),类三为f(f(u)),类四为f(f(f(u)))。这样的分类看起来好像并不困难。但关键是查询是二维的，并没有一个确定的u。接下来就是这个算法的巧妙之处了。

利用递归的LCA过程。当lca(u)执行完毕后，以u为根的子树已经全部并为了一个集合。而一个lca的内部实际上做了的事就是对其子结点，依 此调用lca.当v1(第一个子结点)被lca，正在处理v2的时候，以v1为根的子树＋u同在一个集合里，f(u)+编号比u小的u的兄弟的子树　同在 一个集合里，f(f(u))　+　编号比f(u)小的　f(u)的兄弟　的子树　同在一个集合里.....　而这些集合，对于v2的LCA都是不同的。因此只要 查询x在哪一个集合里，就能知道LCA(v2,x)

还有一种可能，x不在任何集合里。当他是v2的儿子，v3,v4等子树或编号比 u 大的 u 的兄弟的子树（等等）时，就会发生这种情况。即还没有被处 理。还没有处理过的怎么办？把一个查询(x1,x2)往查询列表里添加两次，一次添加到x1的列表里，一次添加到x2的列表里，如果在做x1的时候发现 x2已经被处理了，那就接受这个询问。（两次中必定只有一次询问被接受）

其他介绍:
首先，Tarjan算法是一种离线算法，也就是说，它要首先读入所有的询问（求一次LCA叫做一次询问），然后并不一定按照原来的顺序处理这些询 问。而打乱这个顺序正是这个算法的巧妙之处。看完下文，你便会发现，如果偏要按原来的顺序处理询问，Tarjan算法将无法进行。　　Tarjan算法是利用并查集来实现的。它按DFS的顺序遍历整棵树。对于每个结点x，它进行以下几步操作:

* 计算当前结点的层号`lv[x]`，并在并查集中建立仅包含x结点的集合，即`root[x]:=x`。
    * 依次处理与该结点关联的询问。

* 递归处理x的所有孩子。

* root[x]:=root[father[x]]（对于根结点来说，它的父结点可以任选一个，反正这是最后一步操作了）。

现在我们来观察正在处理与x结点关联的询问时并查集的情况。由于一个结点处理完毕后，它就被归到其父结点所在的集合，所以在已经处理过的结点中（包括 x本身），x结点本身构成了与x的LCA是x的集合，x结点的父结点及以x的所有已处理的兄弟结点为根的子树构成了与x的LCA是father[x]的集合，x结点的父结点的父结点及以x的父结点的所有已处理的兄弟结点为根的子树构成了与x的LCA是father[father[x]]的集合.....（上面这 几句话如果看着别扭，就分析一下句子成分，也可参照右面的图）假设有一个询问(x,y)（y是已处理的结点），在并查集中查到y所属集合的根是z，那么z 就是x和y的LCA，x到y的路径长度就是lv[x]+lv[y]-lv[z]*2。累加所有经过的路径长度就得到答案。　　现在还有一个问题：上面提到的询问(x,y)中，y是已处理过的结点。那么，如果y尚未处理怎么办？其实很简单，只要在询问列表中加入两个询问(x, y)、(y,x)，那么就可以保证这两个询问有且仅有一个被处理了（暂时无法处理的那个就pass掉）。而形如(x,x)的询问则根本不必存储。　　如果在并查集的实现中使用路径压缩等优化措施，一次查询的复杂度将可以认为是常数级的，整个算法也就是线性的了。

附伪代码:

```
LCA(u)
{
   Make-Set(u)
   ancestor[Find-Set(u)]=u
   对于u的每一个孩子v
   {
      LCA(v)
      Union(u)
      ancestor[Find-Set(u)]=u
   }
   checked[u]=true
   对于每个(u,v)属于P
   {
      if checked[v]=true
     then {
         回答u和v的最近公共祖先为 ancestor[Find-Set(v)]
      }
   }
}
```

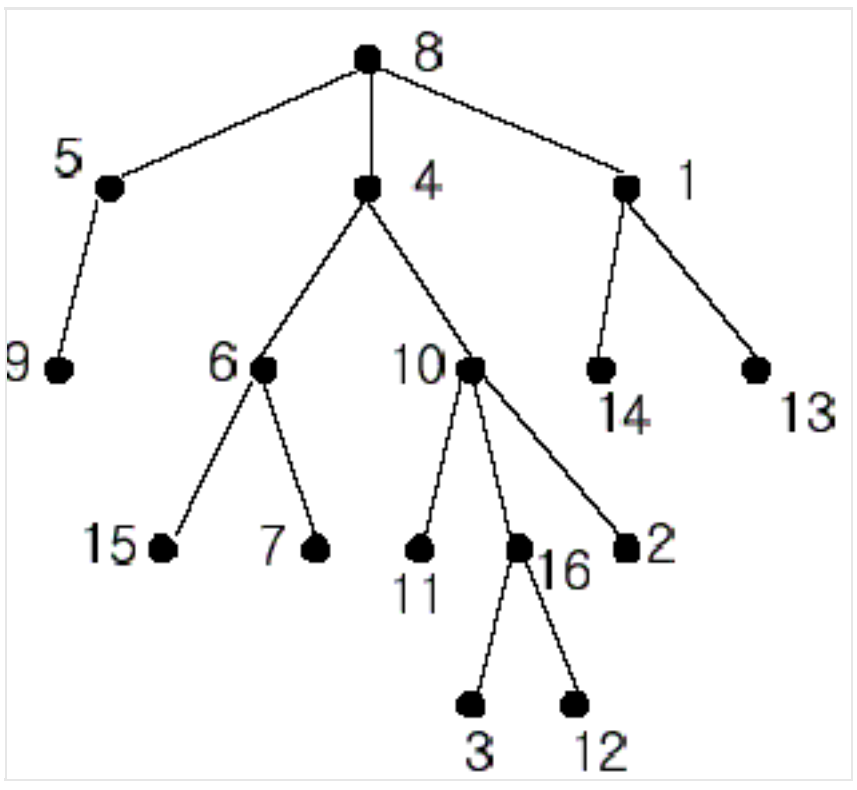
其中，makest就是建立一个集合，makeset（u ）就是建立一个只含U的集合。

findset(u)是求跟U一个集合的一个代表，一般此集合用并查集表示，也就是当前树的root节点。

union（）就是把 V节点生成的子树并入U中。

ancestor就是找跟节点，一直往上找，直至某节点的父节点是自己为止。

这样可能大家看不明白，最好的方法就是大家画个树，模拟一下，就会明白了，主要是那个dfs的尾部递归




```
9 13
8
Press any key to continue
```

感谢以下参考：

http://poj.org/problem?id=1330

http://apps.hi.baidu.com/share/detail/16279376

http://kmplayer.iteye.com/blog/604518

http://blog.csdn.net/lixiandejian/article/details/6661074