最近公共祖先（LCA)

一：定义：

若两个节点的祖先相同，则叫做两个节点的 **公共祖先。**

最近公共祖先(Lowest Common Ancestor)，简称`LCA`，两个节点的最近公共祖先，就是这两个点的公共祖先里面，距离两个节点最近的公共祖先。

二：存图（树）

树： 由`n`个点`n - 1`条边组成的简单图。 树上两点之间有且只有一条`简单路径`。

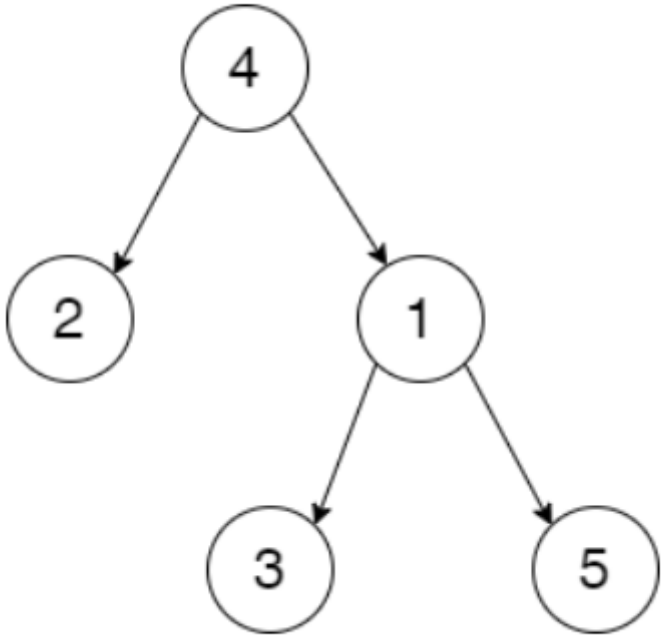
边的类别：

- 有向边：即两个点之间的边是有方向的，可能是`4 -> 3`，表示`4`号点到`3`号点之间有一条有向边，而`3`号点到`4`号点之间是没有边的。也就是所`4`号点可以到达`3`号点，`3`号点不可以到达`4`号点。
- 无向边：即两个点之间的边是没有方向的，或者说两个点之间有两条有向边，即两两之间可以相互到达。`4 -> 3，3 -> 4`这种。可以互相到达，在图论中以`4 — 3`没有箭头的横线表示无向边。

存图方式：

- 邻接矩阵：就是用一个二维数组，`g[i][j] = 1`表示`第 i 个点 和第 j 个点 之间有一条边`，如果`g[i][j] = 0`表示`第 i 个点 和第 j 个点 之间没有边`。

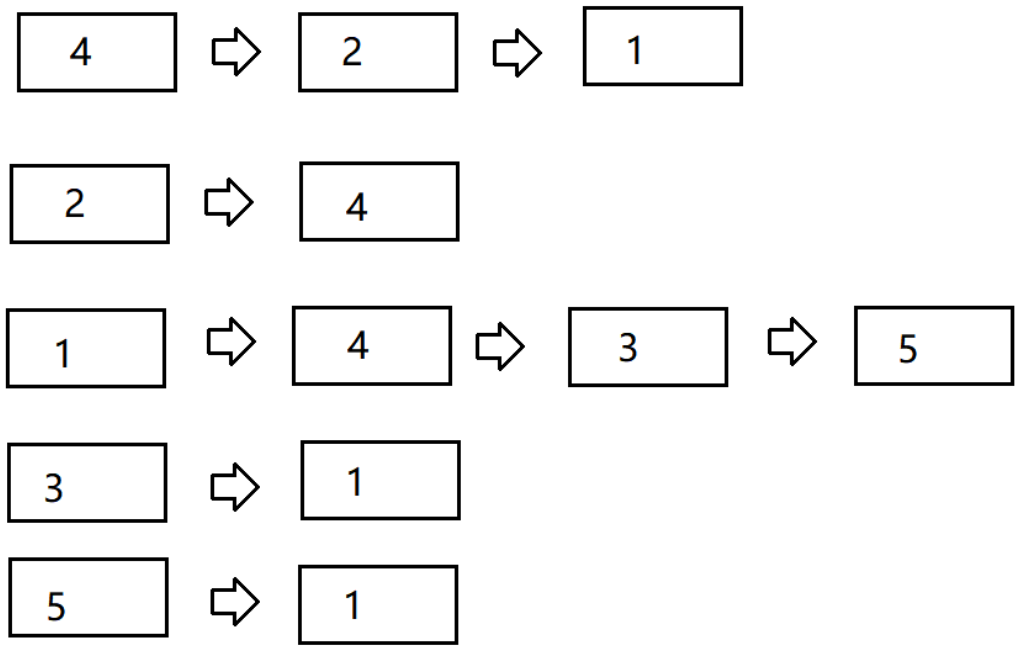
例如：



我们可以看出`4–2 4–1 1–3 1–5`四条边。即

节点	1	2	3	4	5
1	0	0	1	1	1
2	0	0	0	1	0
3	1	0	0	0	0
4	1	1	0	0	0
5	1	0	0	0	0

- 我们发现，其实在上述图中，由很多点之间的边都没有用到，那么我们完全没有必要开这样的空间。所以，邻接矩阵只适用于`点少边多`的情况，即`稠密图`。
- 邻接表：为了解决上述问题，我们用邻接表优化，我们只需要存出现的点即可。我们对于每一个点开一个链表，例如`4`号点，我们有两个边，那么样子就应该是`4 ->2->1 中间顺序是无所谓的`。例如：



- 对于某一个点，我们可以开一个`vector`，后面的点我们可以使用`push_back`进去即可。例如`vector<int> edges[6]  edges[4].push_back(2)  这样就实现了 4 -> 2 在push_back(1) 就实现了4 -> 2 -> 1`
- 链式前向星：并不是本节课内容，而且其实没什么必要，可以用`vector`代替，这里就先不讲了，甘心去的可以自行了解。

树的遍历：

用`dfs`遍历。

模板：

```
void dfs(int u, int f){ // u表示当前节点  f表示当前节点的父亲节点
    for(auto x : edges[u]){ // 遍历邻接表中 u号节点连接的所有点
        if(x == f){ // 由于是双向边，可以会存在递归回去的可能，即父亲到儿子，儿子到父亲，所有如果是父亲就不用递归了
            continue;
        }
        dfs(x, u); // 继续递归，此时儿子是x 父亲是u
    }
}
```

三：性质：

为了方便，我们记某点集 $S = v_1, v_2, v_3, \ldots, v_n$的最近公共祖先为$LCA(v_1, v_2, \ldots, v_n)$或$LCA(S)$。

1. 两个点的`LCA`有且只有一个，并且一定是两个节点到根路径中重复部分最下端的点。
2. $LCA_U = U$
3. u是v的祖先，当且仅当$LCA_{u,v} = u$
4. 前序遍历中，`LCA(S)`出现在所有`S`中元素之前。
5. $d_{u,v} = h_u + h_v - 2 \times h_{lca_{u,v}}$，其中`d`是树上两点距离，`h`是某点到根的距离。

四：求法：

方法一： 朴素求LCA

可以每次找深度比较大的那个点，让它向上跳。显然在树上，这两个点最后一定会相遇，相遇的位置就是想要求的`LCA`。

朴素算法预处理时需要`dfs`整棵树，时间复杂度为$O(n)$，**单次查询**时间$O(n)$。但由于随机树的高为$O(logn)$，所以朴素算法在随机树上的单次查询为$O(logn)$

核心代码：`deep数组表示深度` `fa数组表示某个点的父亲`

```
int lca(int u, int v){
    if(deep[u] < deep[v]){ // 始终保证 u 的深度 比 v 的深度大
        swap(u, v);
    }
    int t = deep[u] - deep[v]; // 他们之间差的深度是 t
    while(t -- ){ // 暴力往上条 t 次
        u = fa[u];
    }
    while(u != v){ // 如果不一样，说明没有相遇，那就继续跳
        u = fa[u];
        v = fa[v];
    }
    return u;
}
```

方法二：倍增求LCA

本算法是对算法一中一步步走的改进，核心实质是让两个结点 **每次向上走 2的幂次方步。**

具体操作如下：

- **第一步，求出倍增数组**：`fa[i][j]`表示第`i`个节点的第2^j次方个父亲。也就是`i`向上走2^j步能够走到的节点。

  我们规定根结点的父亲是它自己，这样根结点往上走还是在根结点。

  对于`j = 0 fa[i][j] 就是 i 的父亲`。

  对于`j > 0 fa[i][j] = fa[fa[i][j - 1]][j - 1] 即节点 i 往上走` 2^{j-1}步后再跳2^{j-1},比如跳四步也就是说2^2等价于先跳两步2^{2-1}再跳两部2^{2-1}。

- **第二步,把两个点移动到统一深度**：还是一样的，我们可以求出两个点深度的差值，但是我们跳跃的时候是跳2的幂次方，例如数字11的二进制表示是`1011`即$2^0+2^1+2^3$所以我们可以发现，如果节点二进制表示下第`j`位是`1`的话，我们就可以跳2^j步。

- **第三步：求出`LCA`**：我们知道二进制的性质，$2^n > 2^{n-1}+2^{n-2}+...+2^0$，我们要跳到他俩不相等的第一个地方，因为我们不能跳过他的`LCA`，不然就不知道`LCA`是谁了。假设`u,v`的`LCA`需要往上跳`L`步，意味着往上跳`L-1`步的时候他们是不相同的，最后向上跳`1`步就是`LCA`。即我们可以按大到小枚举跳2^j步，如果`u,v`向上跳2^i次步为同一个点，就停止，否则就往上跳。最后`LCA`就是`fa[u][0]`，向上跳一步即可。因为前面跳了`L-1`步，差`1`步。

倍增算法的预处理时间复杂度为$O(nlogn)$，**单次查询**的时间复杂度$O(logn)$。

核心代码：

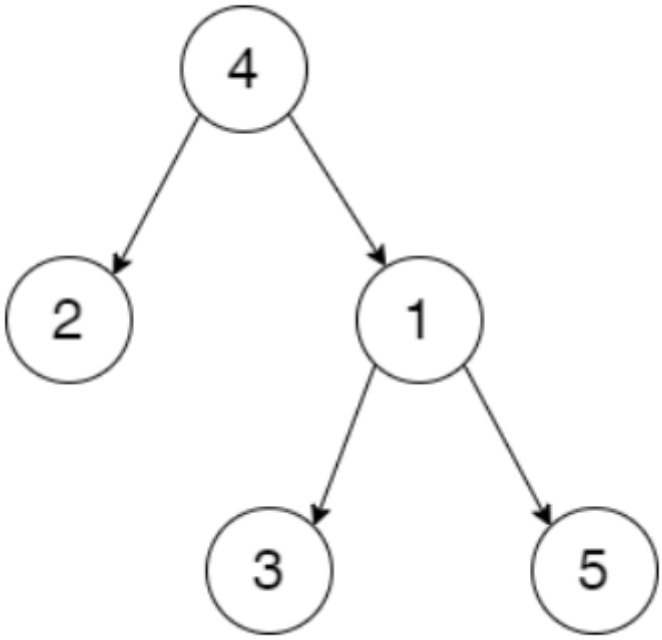
```
void init_Lca(){ // 初始化
    for(int i = 1; i <= 20; i ++ ){
        for(int j = 1; j <= n; j ++ ){
            if(fa[j][i - 1]){
                fa[j][i] = fa[fa[j][i - 1]][i - 1];
            }
        }
    }
}

// 求LCA
int lca(int u, int v){
    if(deep[u] < deep[v]){
        swap(u, v);
    }
    int t = deep[u] - deep[v];
    for(int i = 0; i <= 20; i ++ ){ // 按二进制跳
        if( (t >> i) & 1 ){
            u = fa[u][i];
        }
    }
    if(u == v){ // 如果跳完已经一样了那么就是LCA了
        return u;
    }
    for(int i = 20; i >= 0; i -- ){ // 从大到小开始跳
        if(fa[u][i] != fa[v][i]){ // 只要不同就跳
            u = fa[u][i];
            v = fa[v][i];
        }
    }
    return fa[u][0];
}
```

方法三：Tarjan 求LCA

Tarjan是一个离线求LCA的算法。

所谓的离线，就是要先把所有的询问预处理出来。

具体怎么求呢？我们会一边DFS一边求LCA。



前面提到，某个点的祖先在DFS中一定会在某个点之前被遍历到，比如遍历到3号点的时候，2,4,1都是已经被遍历的，只有5没有被遍历，我们发现，如果想要走到3号来必定要经过他的父亲，所以LCA就会是他的父亲。在举例，我们求3,5的LCA的时候，如果遍历到了5，那么想要到5的一定会经过1，所以3 - 5的LCA就一定是1，即自己的父亲。维护自己的父亲我们应该怎么维护？很自然的想到并查集。

也只有我们经过的点才可以求LCA，例如只遍历到2我们是没办法求2和1 3 5的LCA，因为我们还没遍历到，所以要准备一个vis数组表示是否被遍历到。如果两个点都被遍历到才可以求，此时的LCA(u,v)就是find(u)。

这个标记也应该在dfs的外面标记，我们要保证祖宗应该是最后一个被取消标记的，不然会出现4和5的LCA是1的情况，所以必须先标记5，后标记4。

注意：我们存点的时候，3,5和5,3都要存进询问，因为由于DFS的特殊性，可能遍历到3的时候5并没有被遍历到。

并且合并的时候要在递归自己的儿子之后合并进并查集，因为如果先合并的话，儿子的父亲就不会是父亲了，而是祖宗，例如3和5的祖宗都是1，只有刚刚递归到3结束后在将fa[3] = 1才是将3的祖宗指向了父亲，如果fa[1] = 4了，然后fa[3] = 1 那么fa[3]就会指向4，答案就错误了这个是路径压缩并查集要注意的地方。

复杂度$O(n+q)$

核心代码：que是询问的数组 ans是答案数组

```
```

```cpp
void Tarjan(int u, int f){
    for(auto x : edges[u]){
        if(x == f){
            continue;
        }
        Tarjan(x, u);
        merge(x, u); // 先DFS 后合并
    }
    vis[u] = true; // 先标记儿子，后标记祖宗，因为最后一次DFS出来一定先是儿子。
    for(auto x : que[u]){
        if(vis[x.first]){
            ans[x.second] = find(x.first);
        }
    }
}
```