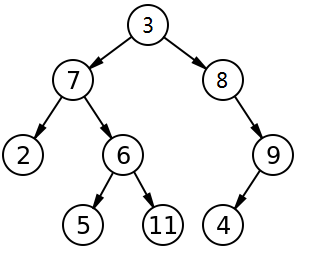
基于欧拉环游树的动态树实现

冼臧越洋 张哲恺 胥拿云 龙思杉

**欧拉环游树**(ETT)：

DFS序：

（图1）

DFS序就是按先序遍历的方式遍历一棵树，例如图（1）的DFS序为：

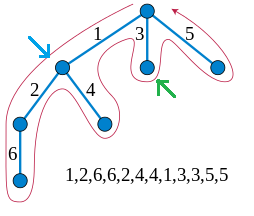
3 7 2 6 8 11 5 9 4

可以发现，任意一颗子树的所有点均在一段连续的区间上，这样就可以方便的维护子树的信息。利用维护子树大小甚至可以支持换父亲的操作。

然而它不能很好的支持换根操作，例如在上图中我们把9换成根，那么DFS序应为：

9 4 8 3 7 2 6 5 11

它和原来的DFS序关系不是很明显（其实是有的），为了能够很好的支持换根，我们掏出了ETT：

图（2）

维护边的ETT比较自然，所以我们先用边的ETT做例子。对于图（2）的树，我们将每条无向边拆成两条有向边，然后以DFS遍历树：

1 2 6 6 2 4 4 1 3 3 5 5

这称为欧拉环游序，显然依旧有一颗子树的边在一段连续的区间上这个性质。

同时我们发现这样做其实是把树变成的一个环，那么对于换根操作无非就是将这个环的起点挪一挪，例如我们将根转为蓝箭头指向的点，那么欧拉环游序变为：

1 3 3 5 5 | 1 2 6 6 2 4 4

注意到从“|”处交换两段序列的位置既得原序列，反之亦然。这样我们就很好的支持了换根。

它也能支持换父亲操作，例如把蓝箭头所指的子树切下来（就是把父亲换成空），我们只需找到它的两条出入边（就是1），然后从原序列切下来即可（此时边1被销毁）。序列变为：

2 6 6 2 4 4

3 3 5 5

如果要把它连到绿箭头所指的点上，我们只要找到这个点的入边，然后插在这条边的后面即可（需要加条边，譬如7）。

3 7 (2 6 6 2 4 4) 7 3 5 5

这里的7是由于加了条边先夹在了3中间，（）实际找到的边是7。

同时我们也能做维护点的ETT（只是有点奇怪），例如对于图（1），它的点ETT序列为：

3 7 2 2 7 6 5 5 6 11 11 6 7 3 8 9 4 4 9 8 (3)

其实它和边的ETT没有什么区别，无非是边变成了用两个点描述而已，一个是起点一个是终点。如果把这个序列首尾相连形成环就能发现它就是边的ETT。

当然它也能换根换父亲，具体不在详细介绍。

如果我们不用换根只有换父亲，那么点ETT序列可以简化成：

3 7 2 2 6 5 5 11 11 6 7 8 9 4 4 9 8 3 (\*)

任意一点为根的子树即为它在序列对应的两点之间的部分，相对于上一个序列它简化的多然而换根不能。

由于对ETT序列的操作就是区间切割，区间插入，区间修改，所以我们使用splay来维护它。所以对任意一个操作它的复杂度为O(logN)。

**Link-cut-tree**

它不是我们的重点，所以就不细讲了（反正你们都懂）。我们知道它可以维护链上的信息，并且preferred child的切换次数是均摊O(logN)的。

**ETT+LCT**

如果我们想同时维护链操作和子树操作怎么办？显然不是每条链在ETT上都是连续的，用LCT维护子树听着更蛋疼（然而貌似top-tree就是这么干的）。难道链用LCT，子树用ETT？然而并不能把信息合并，所以这样做是不行的。

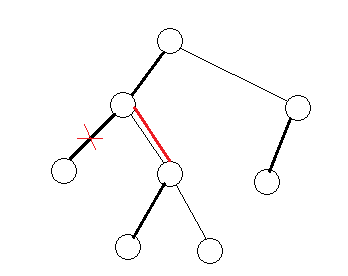
但注意到链其实就是一个特定的DFS序，或者说点人话，ETT中最左边和最右边的一段其实就是一条链，例如(\*)中最左边的3 7 2和最右边的4 9 8 3。这就给了我们一个启发：如果我们能保证每个点的preferred child都是第一个遍历的点（或者形象点，最左边的儿子），那么链上的点也能在ETT上形成连续的区间，也就能高效地修改了。

对于不用换根的树来说，这比较的简单。每次LCT的access中preferred child变化的时候（例如现在7的preferred child变为了6），在ETT中把preferred child对应的子树切下来移到它新父亲的右边即可，对于例子来说就是：

3 7 2 2 | 6 5 5 11 11 6 | 7 8 9 4 4 9 8 3

-> 3 7 | 6 5 5 11 11 6 | 2 2 7 8 9 4 4 9 8 3

这样3 7 6就变成了一段连续区间了。



证明十分简单，只要原先每个preferred child是在最左边，那么将每个变化的preferred child 移到最左边那么性质依旧满足。

例如上图，链上其他重边已经是最左边的边了，只要把红边移到最左边即可。

因为preferred child变化次数均摊O(logN)，每次区间操作是O(logN)的，所以复杂度为O(log2N)的。

算法局限：对于换根的问题

LCT和ETT都支持换根，看起来这个算法能轻松愉悦支持换根吧。然而根本不是这样的好不啦。LCT换根后新根到旧根的链需要翻转，如果我们暴力的在ETT上做是什么样的呢？例如1-2-3-4-5的树，根为1，现在转成5：

|

6-7-8

先access(5) ETT序列:1 2 3 4 5 5 4 3 6 7 8 8 7 6 2 1

然后愉快的翻转，正常来说ETT序列应该是:

5 4 3 2 1 1 2 6 7 8 8 7 6 3 4 5

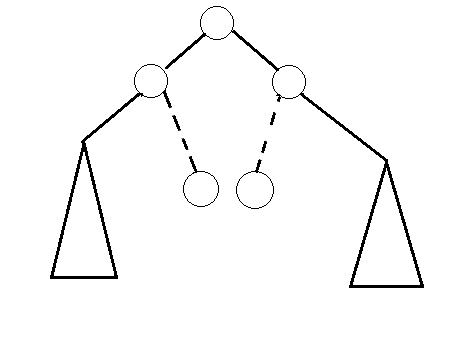
或者是

5 4 3 6 7 8 8 7 6 2 1 1 2 3 4 5

那么这是怎么做的呢，把5切出来，把4切出来放到5里，把3切出来放到4里。。。。。。这个复杂度就变成了链长度\*log(N)，瞬间爆炸。

**算法可能的改进**：

问题显然出在了pushdown翻转标记的时候，那么我们在pushdown的时候同时改变ETT的形态就可以了。例如



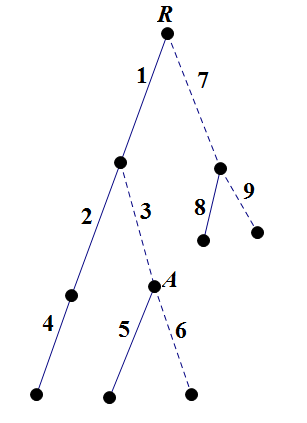
这是一颗辅助树，当根的反转标记下传时，我们若能知道这棵树左子树最右边的点A，根B，右子树最左边的点C，那么在ETT序列中：

..........ABC........CBA........

把C切出来，把B插进C右边，把A插在B右边，把C插回去，就完成了形态的同步。

这样要在辅助树上多维护一些信息，或者暴力的去做，或者用ZZK的SPLAY的迭代器的前驱后继，复杂度大概要再多个logN（算上常数快赶上O(N)了）

**另一种思路**：

如图所示，图中的实边表示LCT中的实边，此时ETT的边序列为

1,2,4,4,2,3,5,5,6,6,3,1,7,8,8,9,9,7

考虑将点A提成根，首先在LCT中access点A，则ETT序列变为

1,3,5,5,6,6,3,2,4,4,2,1,7,8,8,9,9,7

然后在ETT上提根，变为

3,2,4,4,2,1,7,8,8,9,9,7,1,3,5,5,6,6

如果我们把两个3之间的序列翻转，则变成

3,1,7,9,9,8,8,7,1,2,4,4,2,3,5,5,6,6

不难发现，原来的A到R这条链上的边在ETT上依然是连续的，同时这条链上其它子树的序列全部被翻转。

既然被翻转了，那么如果能在需要的时候将这些序列翻转回来就可以了，于是我们得到一种新的思路，提根时将新的根和旧的根之间的点上打上标记，access时如果发现上面的点上有标记，就将其除了preferred child以外的点的子树翻转，同时去掉标记即可。利用Splay可以实现链上打标记和区间翻转。所需要做的是在每次切换preferred child时额外花O(logN)的时间进行区间翻转。操作的次数上面提到是均摊O(logN)的，这样的时间复杂度依然在O(Nlog^2N)

实践表明这样的做法是可行的（见github上的experiment分支），但是运行速度并不理想，在应对10000个点的数据时甚至比暴力慢许多。