浅谈 Link Cut Tree 实现及应用

by MT Chan
修改by Zory

Link Cut Tree，是用来解决**动态树**（Dynamic Tree Problems）问题的。那么，我们先来看看什么是动态树问题（一般都是**无根树**），操作有好多种：

Link(x,y)	**新建**连接 x 和y 的边
Cut(x,y)	**删除**连接 x 和y 的边
Query_Max(x,y)	询问 x 到y 的路径上的**最大点权值**
Query_Sum(x,y)	询问 x 到y 的路径上的**点权值总和**
Add(x,y,k)	x 到y 的路径上的**点权值** 全部**加上 k**
Connection(x,y)	询问 x 和y 是否连通
等等……	

为了解决这个问题，我们伟大的 **Robert Endre Tarjan**（就是那个发明了 **LCA 的 Tarjan 算法**、**强联通分量的 Tarjan 算法**、**RMQ 的 ST 算法**、**Splay Tree** 等等的牛人）和他的一个小伙伴发明了 **Link Cut Tree** 这个**数据结构。**

Link Cut Tree，顾名思义，就是能**支持 Link 操作**和**Cut 操作**的**树型数据结构。**

------------------------**我是分割线**------------------------

其实这个算法和**轻重路径剖分（树链剖分）**是类似的（想学树链剖分可以找 lcd 大神），可以说是**动态版**的**树链剖分**，但因为**轻重路径剖分**只能解决**静态树问题，**而**动态树问题**是会**改变树的结构。**所以，我们要使用**更灵活**的 **Splay** 来取代**线段树**对路径进行维护。

Splay的特点：

①这是一颗**二叉排序树。**一个点的关键值，**大于左子树**所有点的关键值，**小于右子树**所有点的关键值（区分一下**孩子**和**子树**，孩子指的是连接的那个**点**，子树是隶属于孩子的整棵**树**）。

②这是一颗**平衡二叉树。树的层数**会尽量小，从而各种操作的时间复杂度会尽量小（树的操作一般和层数有关）。

③当添加或删除一个点时，有可能会导致变成**非平衡二叉树**，但伸展树

可以经过**旋转（Splay）**调整，维持**平衡二叉树**状态。

其中③正是我们选用它的理由，因为动态树问题，是有**添加**和**删除树上的边**的操作的。

------------------------**我是分割线**------------------------

这个算法的精髓就在于一个函数：**Access ，**这个要重点讲一讲。

（**Access(x)**我们称为**访问点 x**）

其实，这个函数是很简单的，首先我们要了解一些概念（这些跟树链剖分不一样，不要弄混了）：

偏爱子节点（Preferred Child）：如果**最近**一个被访问的点，在 y（y 为 x 的孩子）所在的子树内，那么 y 就是 x 的偏爱子节点（每个点只能有**一**个偏爱子节点）**。**

偏爱边（Preferred Edge）：如果 y 是 x 的孩子而且也是偏爱子节点，那么 x 连到 y 的边为偏爱边。

偏爱路径（Preferred Path）：由**若干**条偏爱边组成的不可再延伸的路径为偏爱路径（一颗树会有**若干**条）**。**

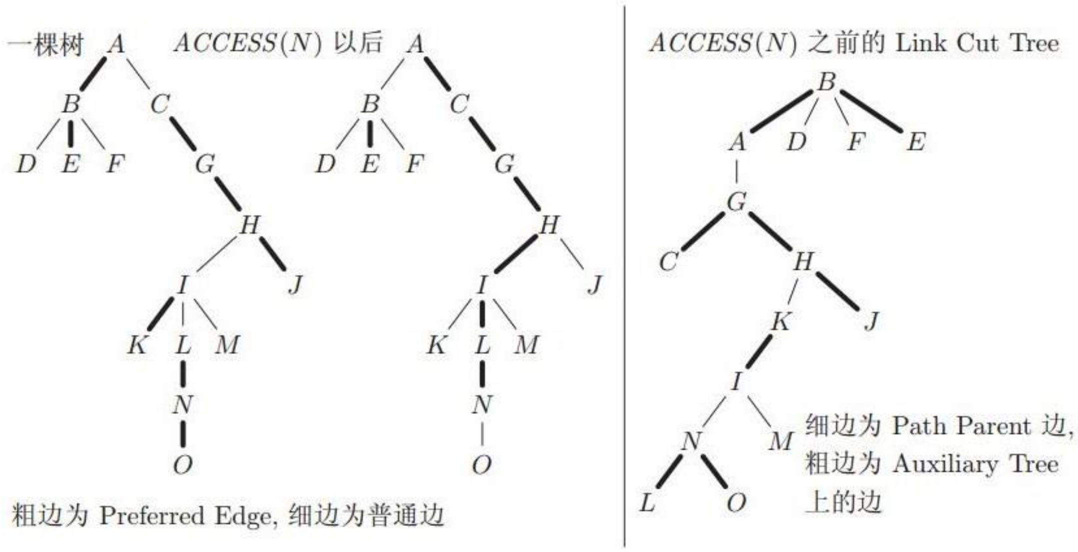
偏爱路径父亲（Path Parent）：偏爱路径最顶端（层数最小） 的点的父亲，就为这条路径的偏爱路径父亲。

偏爱路径父亲边（Path Parent Edge）：**偏爱路径**连到它的**偏爱路径父亲**的那条边，就为这条的**偏爱路径**的偏爱路径父亲边。

对于每一条**偏爱路径**，我们用一棵**伸展树**来维护，这棵伸展树对应这条**偏爱路径**，构建这棵伸展树的**关键值**为**点在树中的深度。**我们称这棵伸展树为**辅助树（Auxiliary Tree）**，并且用**偏爱路径父亲边**把这若干棵辅助树连在一起，形成 **Link Cut Tree。**

------------------------**我是分割线**------------------------

<树 1> **<树 2>** **<树 3>**



（引用并修改 **Yang Zhe 大神**的图）

在**<树 1>**中，K 原本是 I 的偏爱子节点，但是在Access(N)后**<树2>**，K 就

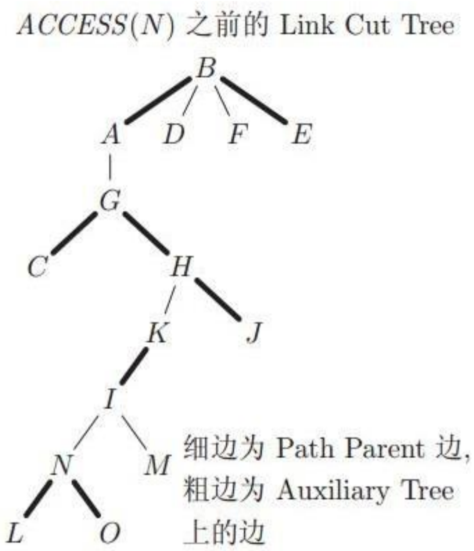
不是 I 的偏爱子节点了，因为 L 所在的子树包含了最近访问的 N，所以 L 取代了 K 变成了 I 的偏爱子节点（**每一个点最多只能有一个偏爱子节点**），其他也同理。

在**<树 3>**中，我们可以分出**若干**棵辅助树。**A<-B->E**就是其中一棵，对应**<树 1>**中的**A->B->E**。由于伸展树的特点②，所以**B**为这棵伸展树的**根**。

在**<树 3>**中，我们可以发现，有许多条**偏爱路径父亲边**把这**若干**棵辅助树连在了一起，构成了 **Link Cut Tree。**

-------------------------**我是分割线**-------------------------

为了方便代码的编写，我们用同一个数组 fa，来保存**偏爱路径父亲**和 **Splay 中的父亲**。那我们怎么用一个数组保存两个信息呢，还是来看这幅图，我们可以发现，对于一个节点：



如果它**不为辅助树的根**（例如 A、C、I）那么它肯定在辅助树中**有父亲**，所以这时我们用 fa 数组保存该节点在 **Splay 中的父亲。**

如果它**为辅助树的根**（例如 B、G、K），那么它肯定在辅助树中**没有父亲**，所以这时我们用 fa 数组保存**偏爱路径父亲**。因为一棵辅助树**对应**一条偏爱路径，所以同一棵辅助树内偏爱路径父亲是相同的。

-------------------------**我是分割线**-------------------------

那我们就可以**愉快地**编写 Access 函数了。

假如要执行 Access(x)，我们**只要保留 x 到根**的信息，所以我们要把 x 的偏爱子节点**断开**，然后**取代**它父亲的偏爱子节点，再把父亲进行**同样的**操作。依此重复，直到父亲不存在（**只有根没有父亲**）。

其实这**断开**和**取代**是类似的，**断开**可认为用空节点**取代。**

我们先把 x 旋转到 Splay 的根上，再用上一个操作的点（**一开始为空**）**取代**它的右孩子（右孩子**层数比 x 大**，所以是**偏爱子节点**），再把 x 设为父亲，重复操作。

Code:

```
void Access(int x)
{
    int last=0;
    while(x!=0)
```

```
    {
        Splay(x);son[x][1]=last;
        Push_Up(x);//孩子变了，更新一下
        last=x;x=fa[x];
    }
}
```

------------------------**我是分割线**------------------------

除了 Access 这个最基本的操作外，还有一些基本操作要掌握：Link、Cut、Find_Root、Connection。

为了实现这些操作，我们要用到一个非常灵活的操作：**换根**！

Make_Root，把一个点换成所在LCT树的根。这个操作，就是把一棵LCT树的根**换成**另一个节点，那么这棵LCT树的形态会变，所以**部分节点**的**层数改变**了，但是 Splay 的关键值就是层数，我们该怎么解决这个问题呢。比如说我们要把 x 换到根，那么只有 **x 到根路径上**的节点的层数发生了改变（这里的层数**不是指层数的值**，而是**指和别的节点的层数大小关系**）。那我们只要**访问** x，就可以分离出这些点，然后把这个 Splay **翻转** 就可以了。

Split，分离出两个点之间的路径。因为 Access 这个操作是可以**分离出某个点到根的路径**，再配合上这个操作，我们就可以很方便的**分离出两个点之间的路径**，而不用去求 LCA。只要**先把一个点换成LCT根**，**再访问另一个点**，就完成了这个路径的分离。

Link，连接两个点。必定有一个是父亲，另一个是孩子。根据树的结构的一个特殊性，**只有根这个节点没有父亲**，所以，我们必须**先把一个点换成LCT根**，**再把它的父亲指向另一个点。**

Cut，切断两个点。我们**先分离出 x 到 y 的路径**，**以 y 为 Splay 的根**，再**切断 y 与左孩子的联系。**因为只有两个点，所以 x 和 y 在 Splay 里必定相连，又因为**分离路径**时把 x **换成了根**，所以 **x 的层数比 y 小**，一定为 y 的左孩子。

Find_Root，询问一个点所在树的根。我们**先访问 x**，**再求出这棵 Splay 层数最小的点**（也就是最左端的点。把 x 旋转为 Splay 的根，一直往左孩子跳）。

Connection，询问两个点是否联通。我们**询问 x 和 y 所在树的根是否相同**就

可以了。因为**联通的两个点必定在一棵树上**，而且**一个根对应一棵树。**

```
Code:
void Make_Root(int x)   //换根
{
    Access(x);
    Splay(x);
    rev[x]^=true;   //Splay 时下放的标记
}
void Split(int x,int y)   //分离两个点之间的路径
{
    Make_Root(x);
    Access(y);
    Splay(y);   //把一个点旋转到 Splay 的根，方便对整条路径的修改
}
void Link(int x,int y)
{
    Make_Root(x);
    fa[x]=y;
}
void Cut(int x,int y)
{
    Split(x,y);
    fa[x]=0;
    son[y][0]=0;
}
int Find_Root(int x)
{
    Access(x);
    Splay(x);
    while(son[x][0]!=0)
    x=son[x][0]; return x;
```

```
}
bool Connection(int x,int y)
{
    return Find_Root(x)==Find_Root(y);
}
```

------------------------**我是分割线**------------------------

有了 Split 这个操作，路径上的问题简直就是**信手拈来**，只要分离出路径，直接查询 Splay 中根维护的信息就好了。

Code:

```
int Query_Sum(int x,int y)
{
    Split(x,y);
    return sum[y];
}

int Query_Max(int x,int y)
{
    Split(x,y);
    return Max[y];
}
```

------------------------**我是分割线**------------------------

但是对于一棵**有根树**，我们调用 Split 这个操作时，里面的 Make_Root 操作会**改变树的根**，那我们就只能使用最古老的办法，**求 LCA**（最近公共祖先）。

我们**先访问 y**，**再访问 x**，在访问 x 的过程中，会遇到一个点的**偏爱路径父亲**为0，那么这个点就是 x 和y 的LCA。

为什么说这个点（称它为 d）是x 和y 的LCA 呢？因为在访问 y 后，从 y 到根的路径上的所有点，构成了一棵新的辅助树，而这棵辅助树是**唯一经过根节点**的，所以也是**唯一没有偏爱路径父亲边**（也就是为 0）的，所以 d 一定是 y 的祖先，在访问 x 的途中遇到的点，肯定也是 x 的祖先。那么只要证明 d 是**深度最大**的就好了。

当访问 x 时，不断往上跳的过程中，每一次跳到**偏爱路径父亲**时，都会跳到另外一颗辅助树中，同时 x 的**偏爱路径父亲**（称它为 p），也是 p 所在的辅助树中 x 的**最近祖先**（层数最大），所以 d 是x 和y 的LCA。对于查询**x 所在偏爱路径的偏爱路径父亲**，我们只要把 x 旋转到 x 所在Splay的根。此时，**fa[x]**就指向 x 所在偏爱路径的偏爱路径父亲。

Code:

```
int LCA(int x,int y)
{
    Access(y);
    int last=0;  //下面为 Access(x)
    while(x!=0)
    {
        Splay(x);
        if(fa[x]==0) return x; //多了这句
        son[x][1]=last;
        Push_Up(x);
        last=x;
        x=fa[x];
    }
}
```

其实有一种更方便的方法

~~只要**记录树的根**，需要进行特定操作时用 Make_Root 操作**把根换回去**就好了。~~