

# 浅谈 Link Cut Tree 实现及应用

by MT Chan

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) 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。

所以, 我们必须要学会伸展树。没学过的同学或者是忘记了的同学, 可以参考文件夹内的《伸展树入门 byscy》和《伸展树的基本操作与应用-杨思雨》自行学习。

我就随便说一下这个数据结构的特点吧:

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, 有可能会变成非平衡二叉树, 但伸展树可以经过旋转 (Splay) 调整, 保持平衡二叉树状态。

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。

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）。

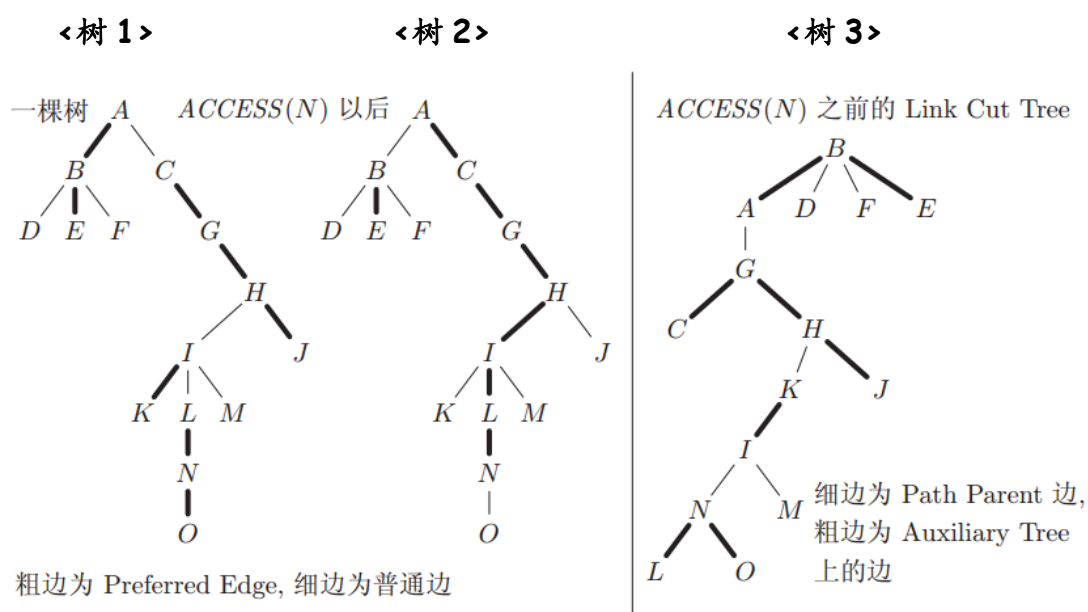
**偏爱边 (Preferred Edge)**：如果 **y** 是 **x** 的孩子而且也是偏爱子节点，那么 **x** 连到 **y** 的边为偏爱边。

**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层数。我们称这棵伸展树为**辅助树 (Auxiliary Tree)**，并且用连到偏爱路径父亲边，把这若干棵辅助树连在一起，形成 **Link Cut Tree**。



(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。

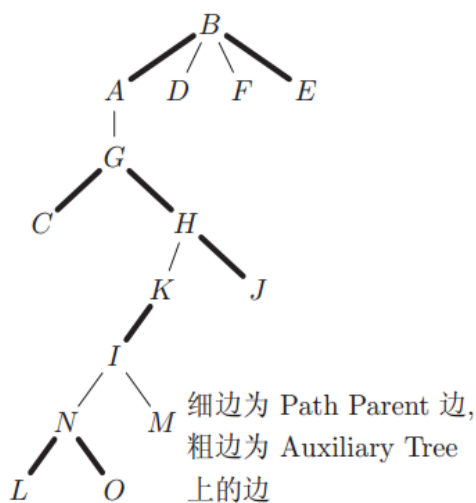
我是分割线

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, 所以同一棵辅助内偏爱路径父亲是相同的。

ACCESS(N) 之前的 Link Cut Tree



细边为 Path Parent 边,  
粗边为 Auxiliary Tree 上的边

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**, 把一个点换成所在树的根。这个操作,就是把一棵树的根换成另一个节点,那么这棵树的形态会变,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** 与右孩子的联系。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)
{
    if(Find_Root(x)==Find_Root(y))return true;
    return false;
}

```

---

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，那我们就可以使用最古老的办法，求 LCA。

LCA，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（最近公共祖先）。

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，构成了一颗新的辅助树，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 是层数最大的就好了。

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];
    }
}

```

---

我是分割线

（其实有一种更方便的方法）

只要记录树的根，需要进行一些特定操作时用 Make\_Root 操作把根换回去

就好了。

---

我是分割线

---

此部分为 **Splay** 的优化，不是必须掌握内容，读者可自行选择学习。

By Yang Zhe

很明显可以看出，我们会反复多次的用到伸展树的操作。虽然伸展树的时间复杂度理论上是很快的，但是由于常数因子太大，往往跑出来的时间不怎么理想，那么，我们可以优化一下其中的 **Splay**、**Rotate**。

**Rotate** 通常的写法是要判断这个点是否为根的，还要判断是父亲的左孩子还是右孩子，这会浪费大量的时间，为了方便这些操作，我们可以定义一些变量。

(以下代码和变量都是对于伸展树而言的)

**type[x]**表示  $x$  是  $x$  的父亲的哪个孩子。**0** 表示是左孩子；**1** 表示是右孩子；**2** 表示  $x$  没有父亲，同时  $x$  也是  $x$  所在的辅助树的根。那么 **Access** 操作时也可以用这个判断一个点是否为一棵辅助树的根。当然，在 **Access** 操作中，把  $x$  设为  $p$  的右孩子时，要更新一下 **type[x]**。

**fa[x]**表示  $x$  的父亲是谁，当  $x$  为  $x$  所在的辅助树的根时，**fa[x]**为 **0**，表示不存在。这个也可以用于判断判断一个点是否为一棵辅助树的根。

**son[x][0]**表示  $x$  的左孩子，**son[x][1]**表示  $x$  的右孩子，分别代替了 **left\_child[x]**和 **right\_child[x]**，目的是方便和 **type[x]**结合使用。

这样就可以实现对 **Splay** 的优化，毕竟聊胜于无。

Code:

```
void Rotate(int x)
{
    int y=fa[x];
    int z=fa[y];
    int a=type[x];
    int b=type[y];
    int g=son[x][!a];
    fa[g]=y;
    type[g]=a;
    son[y][a]=x;
    fa[y]=x;
    type[y]=!a;
    son[x][!a]=y;
    fa[x]=z;
    type[x]=b;
    if(z!=0)son[z][b]=x;
    Push_Up(y); //只用更新 y, x 会在下次 Rotate 时更新
}
```

```
void Splay(int x)
{
    while(true)
    {
        int a=type[x];
        if(a==2)break;
        int y=fa[x];
        int b=type[y];
        if(b==2)
        {
            Rotate(x);
            break;
        }
        if(a==b)Rotate(y);
        else Rotate(x);
        Rotate(x);
    }
    Push_Up(x); //没有下次 Rotate, 所以要在这里更新
}
```

Author: MT Chan

一只七夕还在码论文的单身狗

Blog: [MT\\_Chan.leanote.com](http://MT_Chan.leanote.com)

E-mail: [MT\\_Chan@yeah.net](mailto:MT_Chan@yeah.net)

Tel: 18898489981

Date: 2016.8.9