神奇的 Splay
The Magical Splay

A product of sqybi

关键字 伸展树 Splay

摘要 引言部分介绍了平衡二叉树的一些基本知识。
基本操作部分介绍了 Splay 的两个十分简单的操作，即旋转与伸展操作。它们是其它操作的基础。
进阶操作部分介绍了一些比较常用的操作。
特殊操作部分介绍了一些利用了 Splay 特性或 Splay 特有的操作。
广泛应用部分通过几个例子使读者了解 Splay 的一些很奇特的应用。

说明 本文旨在帮助对平衡二叉树有所了解但不熟悉的 OIer 了解 Splay。

联系方式
E-mail: sqybi@126.com
Blog: http://sqybi.72pines.com/

欢迎大家找到文章中的各类 bug（包括描述错误，各种科学性错误，错别字，伪代码错误，字体错误等各种可能影响阅读的问题）并与本人联系。
另：文章中引用了一些论文、题目等资料，如果你认为本文引用的某些资料侵犯了你的权益，也请与本人联系。

引言

伸展树是平衡二叉树的一种。普通的二叉搜索树在一些特殊数据下会退化成链状，不能保证均摊 O(nlogn)的复杂度，平衡二叉树则通过一些条件下的旋转操作保证了 O(nlogn)的复杂度；而 Splay 则通过其特殊的伸展（Splay）操作保证了均摊复杂度为 O(nlogn)。

两类平衡二叉树 竞赛中经常见到的平衡二叉树有两类，一类是严格维护平衡的，可以保证每次操作的复杂度严格为 O(logn)，例如 AVL，SBT 等；另一类是非严格维护平衡的，但每次操作均摊复杂度为 O(logn)，例如 Treap，Splay 等。虽然第二种平衡二叉树的速度有可能会慢一些，但这并不会影响太多。

Splay 的优势 Splay 作为竞赛中常用的最灵活的平衡二叉树，有着许多先天的优势。而这些优势，都是和 Splay 的灵活分不开的。其中一些虽然 Treap 靠更改附加域也可以做到，但 Splay 的原生支持显然更好用些。

首先，Splay 不需要任何附加空间。它既不需要像 SBT 一样维护 size 域（即使 size 域在大多数情况下对于任何平衡二叉树都很有用，不过不排除某些情况下 size 域是不必要的），也不需要像 Treap 一样维护一个随机数。

其次，Splay 的维护平衡很自然。不需要任何过多的比较，只需要每次访问到某个叶子节点之后将它旋转到根，伸展操作与其他操作一气呵成。

再次，Splay 支持的操作更多。比如删除数据结构中比某个值大的所有数，用其它平衡二叉树只能一个一个删掉，时间复杂度为 O(klogn)；而 Splay 可以在 O(logn)的复杂度完成这个操作。

最后，Splay 能够解决的问题更广泛。在某些问题中，Splay 甚至可以替代块状链表维护一个序列！这也正是本文将它称作“The Magical Splay”的原因。

Splay 的不足 Splay 的不足似乎不是那么明显，除了速度是以上提到的几个平衡二叉树中最慢的一个。但要知道的是，这只是常数上的差异。

也就是说，Splay 虽然可以被构造数据导致速度变得很慢，但均摊复杂度依然是 O(nlogn)。

例如 NOI 2006 的生日快乐（happybirthday），SBT 在最后一个点可以做到 3s 跑过，而 MaShuo 的 Splay 只能做到 6s，我的 Splay 更是用了 8s 才跑过了最后一个点（全部在本机测试）。

程序见 happybirthday_winsty_sbt.dpr，happybirthday_MaShuo_Splay.dpr 和 happybirthday_sqybi_Splay.dpr。

基本操作

参见芜湖一中杨思雨的论文《伸展树的基本操作与应用》和马朔的《伸展树操作详解》。

旋转 无论是什么平衡二叉树，都会用到旋转操作。旋转操作可以使得某一个结点提升到他父亲的位置而不破坏平衡二叉树的性质。
下面是一个很经典的旋转的图示：

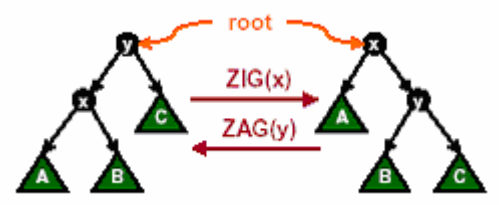

图 1 旋转操作

其中我们将 ZIG 操作称为右旋，而 ZAG 操作称为左旋。另外为了描述方便，对于图中的 ZIG 操作，我们称作将 x 结点右旋（而不是某些论文里提到的将 y 结点右旋，读者可以很容易发现这两种描述是等价的）；同样对于图中的 ZAG，我们称作将 y 结点左旋。
为什么旋转操作不破坏二叉树的性质呢？下面我们来简单的以 ZIG 操作举例说明。
我们用 A，B，C 表示 A，B，C 这三棵子树里所有值组成的集合。那么我们有：

$$\forall a \in A, b \in B, c \in C, \exists a \le x \le b \le y \le c$$

而很容易发现，在 ZIG 操作之后，这种关系仍然满足。

实际上，ZIG 和 ZAG 的思想很简单。以 ZIG 为例，只是将 x 的右子树 B 先暂时拿开，然后将 y 接到 x 的右子树上，x 接到原先 y 的位置上，再把 B 接到 y 的左子树上，ZIG 操作就完成了。程序中实际的实现也是这样做的。

```
01  PROCEDURE LeftRotate(x)
02    y ← Father[x]
03    RightSon[y] ← LeftSon[x]
04    IF (LeftSon[x] <> 0)
05      Father[LeftSon[x]] ← y
06    END IF
07    Father[x] ← Father[y]
08    IF (Father[y] <> 0)
09      IF (y = LeftSon[father[y]])
10        LeftSon[father[y]] ← x
11      ELSE
12        RightSon[father[y]] ← x
13      END IF
14    END IF
15    LeftSon[x] ← y
16    Father[y] ← x
17    Update(x)
```

```
18    Update(y)
19  END PROCEDURE
```

读者可以自行完成 RightRotate(x)操作，而我们可以发现这两种操作很接近。所以，实际实现中为了简便，我们用 Rotate(w, x)代替以上两种旋转操作——w=0 为左旋，w=1 为右旋。同时 son[0, x]替代 LeftSon[x]，son[1, x]替代 RightSon[x]。具体程序见 superbt.dpr。

程序里的 Update(x)操作表示将 x 结点的 size 域更新（size[x]表示 x 这棵子树的结点个数，包括 x 结点，在后面的查找第 k 小的操作中会用到）。更新时只需要 size[x] ←size[LeftSon[x]]+size[RightSon[x]]。为了避免 LeftSon[x]或 RightSon[x]等于 0，我们可以让 size[0] ←0。

伸展操作 伸展操作是 Splay 的精髓，也是 Splay 维护平衡的关键（如果没有 Splay 操作，伸展树就无法保证 O(nlogn)的均摊复杂度）。伸展操作仅仅用到了两种 Rotate 的简单组合——组合出的四种新操作称作 ZIG-ZAG，ZAG-ZIG，ZIG-ZIG 与 ZAG-ZAG，加上原先的 ZIG 和 ZAG，总共六种操作，对应三种不同情况。

考虑需要将 x 结点伸展到根的操作。

第一种情况：如果 x 结点的父亲 y 已经是根结点，我们只需要简单的对它做 ZIG 或 ZAG 的操作，它就会被伸展到根，如图 1 所示。

第二种情况：如果 x 结点的父亲 y 不是根结点，设 y 结点的父亲为 z。若 y 是 z 的左孩子，且 x 是 y 的左孩子，则我们进行 ZIG-ZIG 操作；若 y 是 z 的右孩子，且 x 是 y 的右孩子，则我们进行 ZAG-ZAG 操作。

以 ZIG-ZIG 为例，我们需要做的只是将 y 结点先 ZIG，再将 x 结点 ZIG。很显然这样 x 结点就会替代原先 z 结点的位置。实际上，ZIG-ZIG 操作之后结点的位置变化如图 2：

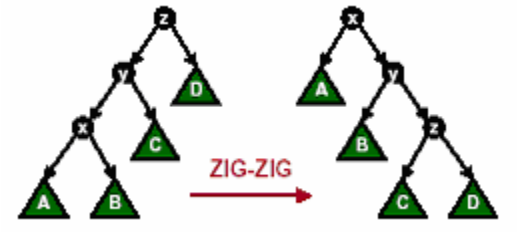

图 2 ZIG-ZIG 操作

ZAG-ZAG 操作类似。

第三种情况：如果 x 结点的父亲 y 不是根结点，设 y 结点的父亲为 z。若 y 是 z 的左孩子，且 x 是 y 的右孩子，则我们进行 ZAG-ZIG 操作；若 y 是 z 的右孩子，且 x 是 y 的左孩子，则我们进行 ZIG-ZAG 操作。

对于 ZIG-ZAG 操作，和 ZIG-ZIG 有所不同——它需要先把 x 结点右旋，再把 y 结点左旋。旋转后结点变化如图 3：

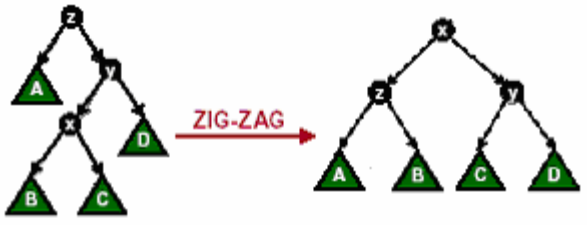

图 3 ZIG-ZAG 操作

或许这些 ZIG 和 ZAG 排列组合出的名字能让你们头昏脑胀，没关系，这是正常的生理反应^_^。你只需要记住以下几点，上面的就都可以忽略了（那我写那些干什么=.=!）。

如果当前结点是根节点的孩子，直接旋转一次即可。

如果当前结点，当前结点的父亲结点，当前结点的父亲结点的父亲结点三点成一直线，先旋转当前结点的父亲结点，再旋转当前结点。

如果当前结点，当前结点的父亲结点，当前结点的父亲结点的父亲结点三点成一折线，先旋转当前结点，再旋转当前结点的父亲结点。

可以根据图 2 和图 3 中的例子自己在草稿纸上模拟一下，印象会更深刻~

```
01  PROCEDURE Splay(x)
02    y ← 0
03    WHILE (father[x] <> y)
04      IF (Father[Father[x]] = y)
05        IF (x = LeftSon[Father[x]])
06          RightRotate(x)
07        ELSE
08          LeftRotate(x)
09        END IF
10      ELSE
11        IF (Father[x] = LeftSon[Father[Father[x]]])
12          IF (x = LeftSon[Father[x]])
13            RightRotate(Father[x])
14            RightRotate(x)
15          ELSE
16            LeftRotate(x)
17            RightRotate(x)
18          END IF
19        ELSE
20          IF (x = RightSon[Father[x]])
21            LeftRotate(Father[x])
22            RightRotate(x)
23          ELSE
24            RightRotate(x)
25            LeftRotate(x)
26          END IF
27        END IF
28      END IF
29    END WHILE
30    IF (y = 0)
31      root ← x
32    END IF
33  END PROCEDURE
```

注意到程序中使用到变量 y，且 y 被赋值 0。y 的含义是，使 x 在 Splay 操作结束后作为 y 的孩子。上面介绍的 Splay 需要将 x 伸展到根结点，伸展后 x 的父亲为 0（即没有父亲），所以 y=0。实际上我们可以使 y 的值为 x 的任意一个祖先（Splay 函数定义为 Splay(x, y)），这样 Splay 操作就更加灵活，可以将 x 伸展到任何我们想要的位置（想想为什么可以做到？）。后面的程序中会用到这种灵活的伸展。

进阶操作

接下来文章会介绍其它平衡二叉树也都会用到的一些经典操作：插入一个数，删除某个特定值，查找某个值在数据结构中的排名（即查询它是第几小的数），查找某个排名的数是多少。对于后两种操作，我们需要维护一个 size 域。而 size 域也可以用于维护重复元素。

插入操作 对于 Splay 来说，插入操作和其它平衡二叉树没有什么太大的区别，唯一的不同就在于插入操作之后需要将刚插入的结点伸展到根。这看起来没有什么必要，但详细的论证表示伸展操作可以降低树的高度，从而保证 O(nlogn)的均摊复杂度。具体的证明见杨思雨的论文《伸展树的基本操作与应用》。
这里以没有重复元素的插入为例。插入时，首先从根节点开始，每次判断需要插入的数据的序号与当前结点的序号。如果需要插入的数据较小，则插入到右子树，否则插入到左子树中。如果需要插入的位置没有结点，那么中止循环，并在此位置插入一个结点。插入结点之后，不要忘记把这个结点伸展到根（有兴趣的话可以尝试一下对任意一个子程序省略这句 Splay 操作，你会惊奇的发现 Splay 的速度慢了 10 倍甚至更多，如果不是随机数据的话；当然对 Insert 操作省略这句话会使 Splay 变得更慢）。

```
01  PROCEDURE Insert(v)
02    x ← root
03    WHILE (true)
04      size[x] ← size[x] + 1
05      IF (v < data[x])
06        IF (LeftSon[x] = 0)
07          BREAK
08        ELSE
09          x ← LeftSon[x]
10        END IF
11      ELSE
12        IF (RightSon[x] = 0)
13          BREAK
14        ELSE
15          x ← RightSon[x]
16        END IF
17      END IF
18    END WHILE
19    TotNode ← TotNode + 1
20    data[TotNode] ← v
21    Father[TotNode] ← x
22    size[TotNode] ← 1
23    IF (v < data[x])
24      LeftSon[x] := TotNode
25    ELSE
26      RightSon[x] := TotNode
27    END IF
```

```
28    Splay(x, 0)
29  END PROCEDURE
```

这个程序并不是最短的写法，但是很容易理解。程序表示插入一个值 v。第 2 行到第 17 行寻找到了当前数需要插入的位置 x，而应该插入在 x 的左边还是右边需要根据 v 和 data[x] 的关系而定。注意每次遍历到 x 的时候要把 x 的 size 加 1，因为如果遍历到了 x，那么 x 一定会是新加入结点的祖先。
建议自己画一棵树模拟一下。

删除操作 Splay 有一种很特别的删除操作，但 Splay 也支持普通平衡二叉树的删除操作，因为 Splay 没有严格维护平衡的条件，所以可以对它进行任何其它平衡二叉树上的操作。但是，Splay 的特殊删除操作更美妙，更简便，更易于理解。所以本文中将介绍这种删除操作。
如果忽略一些繁琐的边界条件，那么删除结点 x 的整个过程可以被描述为这样：
定义 y 为 x 的前驱结点，z 为 x 的后继结点。
首先，把 y 伸展到根；然后，把 z 伸展到根的右子树；这时，x 就是 z 的左孩子。只需要把 x 结点从 z 上脱离（LeftSon[z] ← 0），然后更新 z 和 y 的 size 域即可。
当然，实际应用中还要考虑这样几种情况：x 没有前驱结点，x 没有后继结点，x 既没有前驱结点又没有后继结点。
对于第一种情况，我们只需要把 z 伸展到根然后删除 x；第二种情况类似。而第三种情况，则更加简单——x 一定是树中剩下的唯一一个结点，所以把 root 置为 0 即可。
由于判断过于繁琐，所以伪代码中假定了不会出现 x 没有前驱或后继结点的情况。而实际程序中进行几个判断即可，具体代码见 superbt.dpr（题目见 superbt.doc）。
另外关于前驱和后继结点的求法，见伪代码或后文。

```
01  PROCEDURE Delete(x)
02    Splay(x, 0)
03    y ← LeftSon[x]
04    WHILE (RightSon[y] != 0) //求 x 的前驱：左子树里值最大的一个
05      y ← RightSon[y]
06    END WHILE
07    z ← RightSon[x]
08    WHILE (LeftSon[z] != 0) //求 x 的后继，同上
09      z ← LeftSon[z]
10    END WHILE
11    Splay(y, 0)
12    Splay(z, y)
13    LeftSon[z] ← 0
14    Update(z)
15    Update(y)
16  END PROCEDURE
```

查找某个数 平衡树经典操作。

查找一个数只需要先从根结点找起，分三种情况（以左小右大的存储方式为例）：
如果当前结点的值大于要查找的值，那么继续找左孩子；
如果当前结点的值小于要查找的值，那么继续找右孩子；
如果当前结点的值恰好是要查找的值，那很好，直接返回当前结点编号^_^。
递归进行操作，直到返回结果或者到叶子节点还没有找到（说明要查找的值不存在于平衡树中）。

```
01  FUNCTION Find(v)
02    x ← root
03    WHILE (x != 0)
04      IF (v < data[x])
05        x ← LeftSon[x]
06      ELSE
07        IF (v > data[x])
08          x ← RightSon[x]
09        ELSE
10          BREAK
11        END IF
12      END IF
13    END WHILE
14    Splay(x, 0)
15    RETURN x
16  END FUNCTION
```

查找数的排名 即查询这个数是平衡树中的第几小数。有了 size 域和伸展操作，这个操作变得极其简单。
首先，找到这个数；然后，将它伸展到根；最后，这个数的排名就是这个结点的左子树大小+1。

```
01  FUNCTION Rank(v)
02    x ← Find(v)
03    Splay(x, 0)
04    RETURN size[LeftSon[x]] + 1
05  END FUNCTION
```

查找某排名的数 同样经典的操作（也就是查找第 k 小数的操作），和查找某个数很相像。
从根结点开始递归进行在当前子树查找第 k 小操作：如果当前结点的左子树大小大于等于 k，那么在左子树查找第 k 小的数；如果 k 大于左子树大小加 1，那么在右子树查找第 k-(左子树大小+1)小的数；否则 k 一定等于左子树大小加 1，此时返回当前结点编号。
建议在看程序之前，思考一下 k，左子树大小+1 和 k-(左子树大小+1)分别是什么含意。
此时，用一个例子来模拟仍然是最明智的选择^_^。

```
01  FUNCTION Getk(v)
02    x ← root
03    WHILE (v != size[LeftSon[x]] + 1)
04      IF (v <= size[LeftSon[x]])
05        x ← LeftSon[x]
06      ELSE
07        v ← v - (size[LeftSon[x]] + 1)
08        x ← RightSon[x]
09      END IF
10    END WHILE
11    Splay(x, 0)
12    RETURN x
13  END FUNCTION
```

特殊操作

对于 Splay，除了上述操作，还有一些它特有的操作。下面我们要介绍的就是两种它特有的操作：查找前驱、后继，删除大于某个值的所有数。这两种操作虽然在其它平衡树中也可以完成，但 Splay 在进行的时候充分利用了自身的性质。实际上删除操作也属于此类，不过因为它过于普通，没有归为此类。

前驱后继 这点在前文的 Delete 操作部分已经提到，这里再总结一下。由于前驱后继基本相似，所以以前驱结点为例来介绍。
首先我们可以发现，将结点 x 伸展到根之后，x 的前驱结点是它的左子树中序号最大的一个结点。而这个结点的位置，一定是 x 的左孩子的右孩子的右孩子的……的右孩子。很显然如果 x 的前驱结点不在这个位置，那么一定能够找到比它大的结点，不能满足“x 的前驱结点是它的左子树中序号最大的一个结点”。后继结点相同。这样就可以在 O(logn)的时间复杂度内完成查找前驱/后继的操作。
伪代码见 Delete 部分即可。

删除大于 l 且小于 r 的数 依然和 Delete 部分很像。只不过这次我们需要先将序号小于 l 的最大的结点伸展到根，序号大于 r 的最小的结点伸展到根的右子树。也就是说，将 l 的前驱结点伸展到根，r 的后继结点伸展到根的右子树。此时，r 的后继结点的左子树里，就存放着从 l 到 r 的所有结点。此时只需要把这棵子树和 r 的后继结点断开即可。
这里同样忽略了处理边界情况，边界情况和 Delete 操作基本相同，伪代码里也忽略掉。
此操作的伪代码和 Delete 操作很像。

```
01  PROCEDURE DeleteTree(l, r)
02    Splay(l, 0)
03    y ← LeftSon[l]
04    WHILE (RightSon[y] != 0)
05      y ← RightSon[y]
06    END WHILE
07    Splay(r, 0)
08    z ← RightSon[r]
09    WHILE (LeftSon[z] != 0)
10      z ← LeftSon[z]
11    END WHILE
12    Splay(y, 0)
13    Splay(z, y)
14    LeftSon[z] ← 0
15    Update(z)
16    Update(y)
17  END PROCEDURE
```

广泛应用

引言中提到 Splay 的应用十分广泛，实际也如此。这里介绍它的一个十分特别的应用：在某些情况下替代块状链表。

替代块状链表 对于什么是块状链表，本文不做介绍，有兴趣的读者可以去阅读相关论文（如苏煜的论文《对块状链表的一点研究》）。
块状链表的作用就是维护一个序列。而 Splay 作为一个树状结构，怎么做到这点呢？我们考虑给序列的每个元素加上一个权值，这个值就是元素在序列中的位置。我们可以发现，这时以这个值为关键字将这个序列插入一棵二叉查找树，树的中序遍历竟然就是这个序列！而无论树的形态如何变化，只要没有破坏二叉查找树的性质，我们可以很容易发现树的中序遍历也不会发生变化。所以，伸展操作不会破坏中序遍历，Splay 也就可以用来存储这个序列。这样维护序列的时间复杂度为 O(nlogn)，在部分题目中完全可以替代块状链表（时间复杂度为 O(n*sqrt(n))）。
实际上，我们没有必要加入权值，它除了能够帮助我们理解为什么树状结构可以存储一个序列以外没有任何其它的作用。实现的时候只需要根据序列递归建树即可（每次以序列最中间的元素为树根，左右两部分元素递归建树）。

实际应用 来看一个 Splay 很经典的应用。

题目要求对于一个序列，维护三种操作：删除当前序列第 x 个元素到第 y 个元素（例如序列 ADBCDCE 删除第 2~4 个元素之后变为 ADCE），在当前序列第 x 个元素后插入一段元素（例如 ABCD 在第 2 个元素后插入序列 DEC 变成 ABDECCD），将当前序列的第 x 个元素到第 y 个元素逆序（如 ACEDCBDF 的第 3~7 个元素逆序后变成 ACDBCDEF）。

第一个操作的实现很容易：只需要先把第 x-1 个元素伸展到根，再把第 y+1 个元素伸展到 x 的右子树的根。这样，第 x 个元素到第 y 个元素的序列就会作为第 y+1 个元素的左子树在平衡树里出现，如图 4（和“删除大于 l 小于 r 的数”操作很像，自己用几个例子画画吧 ^_^），删除这棵子树即可。

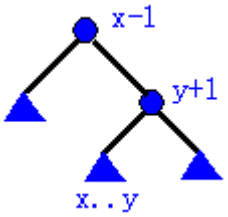

图 4 伸展后树的形态

第二个操作更加简单：先把第 x 个元素伸展到根，再把第 x+1 个元素伸展到 x 的右子树的根。此时再把需要插入的序列建立一棵子树插入到第 x+1 个元素的左子树（原先为空）即可。这个操作和第一个操作很像。
那么，对于第三个操作，我们要怎么办呢？

这里，我们要借鉴线段树的一个经典操作——懒标记。
首先，把第 x-1 个元素伸展到根；然后，将第 y+1 个元素伸展到根的右孩子。此时，第 x 个元素到第 y 个元素组成的序列就出现在第 y+1 个元素的左子树的位置上。此时，对此子

树的根打上一个懒标记（如果已经存在一个懒标记则撤消），说明子树表示的序列需要被翻转，而不必立即进行翻转。当之后访问到这个结点的时候，只需要将懒标记“下放”到两个孩子并交换两个孩子的位置即可。这样做的正确性毋庸置疑，而时间复杂度显然不会增加一个数量级——懒标记的下放只不过在旋转操作中顺便进行罢了。

参考文献及附件

1. 《伸展树的基本操作与应用》，杨思雨，2006
2. 《伸展树操作详解》，马朔，2006
3. 《对块状链表的一点研究》，苏煜，2007
4. superbt，作者未知，时间未知
5. 《生日快乐》，NOI，2006

感谢

感谢 winsty 指出关于 happybirthday 的最后一个点的数据使 Splay 变慢的问题。
感谢 winsty 指出关于 size 域的作用问题。
感谢黄磊指出 Splay 的均摊复杂度不变的问题。