运用伸展树解决数列维护问题

By Crash[1]

【关键词】数列维护问题、伸展树

【摘要】对于数列维护问题，我们常用的一种手段是线段树。但使用线段树有一定的局限性，本文介绍运用伸展树解决这类问题，并且可以实现更多的功能。

【目录】

[1] Blog 地址：http://hi.baidu.com/oimaster

（1）伸展树的伸展操作

伸展树属于一种平衡树，我们可以把其当做普通的排序二叉树使用，但是其功能不局限于此。

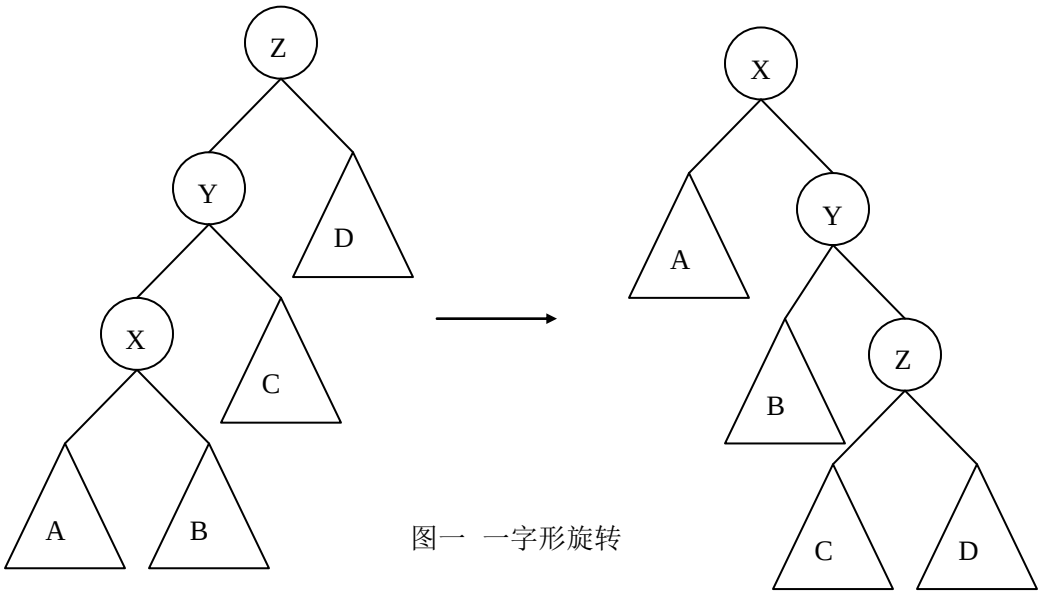
伸展树的核心就是 *Splay*（伸展）操作，$Splay(X)$ 表示把结点 X 旋转到整棵树的根，这里的旋转方法同其他的平衡树。一般人对于这个操作，可能首先想到的方法就是简单的一层一层向上旋转，但是这样常常无法改变树的形态，一条链 *Splay* 后还是一条链。因此伸展树使用了一些新的方法进行这个看似简单的操作。

分三种情况讨论：

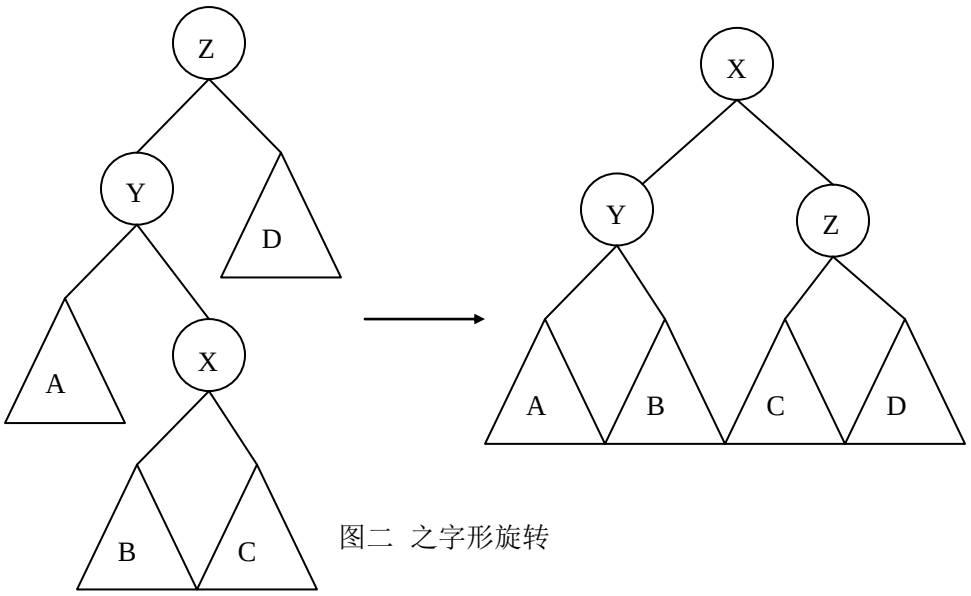
1. 如果当前结点父结点即为根结点，那么我们只需要进行一次简单旋转即可完成任务，我们称这种旋转为单旋转。

2. 设当前结点为 X ， X 的父结点为 Y ， Y 的父结点为 Z ，如果 Y 和 X 同为其父亲的左孩子或右孩子，那么我们先旋转 Y ，再旋转 X 。我们称这种旋转为一字形旋转。

3. 最后一种就是和上面相反的情况，这时我们连续旋转两次 X 。我们称这种旋转为之字形旋转。



图一 一字形旋转



图二 之字形旋转

通常来说，每进行一种操作后都会进行一次 *Splay* 操作，这样可以保证每次操作的平摊时间复杂度是 $O(\log n)$。关于证明可以参见相关书籍和论文。

既然可以把任何一个结点转到根，那么也就可以把任意一个结点转到其到根路径上任何一个结点的下面（特别地，转到根就是转到空结点 *Null* 的下面）。下面的利用伸展树维护数列就要用到将一个结点转到某个结点下面。

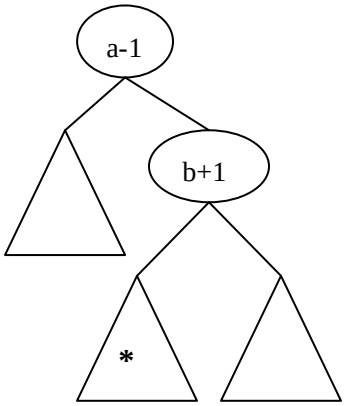
最后附上 Splay 操作的代码：

```
// node 为结点类型，其中 ch[0] 表示左结点指针，ch[1] 表示右结点指针
// pre 表示指向父亲的指针

void Rotate(node *x, int c)  // 旋转操作，c=0 表示左旋，c=1 表示右旋
  {
    node *y = x->pre;
    y->ch[! c] = x->ch[c];
    if (x->ch[c] != Null) x->ch[c]->pre = y;
    x->pre = y->pre;
    if (y->pre != Null)
      if (y->pre->ch[0] == y) y->pre->ch[0] = x; else y->pre->ch[1] = x;
    x->ch[c] = y, y->pre = x;
    if (y == root) root = x;  // root 表示整棵树的根结点
  }

void Splay(node *x, node *f)  // Splay 操作，表示把结点 x 转到结点 f 的下面
  {
    for ( ; x->pre != f; )
      if (x->pre->pre == f)  // 父结点的父亲即为 f，执行单旋转
        if (x->pre->ch[0] == x) Rotate(x, 1); else Rotate(x, 0);
```

```
    else
      {
        node *y = x->pre, *z = y->pre;
        if (z->ch[0] == y)
          if (y->ch[0] == x)
            Rotate(y, 1), Rotate(x, 1);  // 一字形旋转
          else
            Rotate(x, 0), Rotate(x, 1);  // 之字形旋转
        else
          if (y->ch[1] == x)
            Rotate(y, 0), Rotate(x, 0);  // 一字形旋转
          else
            Rotate(x, 1), Rotate(x, 0);  // 之字形旋转
      }
  }
```

（2）在伸展树中对区间进行操作

首先我们认为伸展树的中序遍历即为我们维护的数列，那么很重要的一个操作就是怎么在伸展树中表示任意一个区间。比如我们要提取区间$[a,b]$，那么我们将a前面一个数对应的结点转到树根，将b后面一个结点对应的结点转到树根的右边，那么根右边的左子树就对应了区间$[a,b]$。



图三 提取区间

其中的道理也是很简单的，将a前面一个数对应的结点转到树根后，a及a后面的数就在根的右子树上，然后又将b后面一个结点对应的结点转到树根的右边，那么$[a,b]$这个区间就是图三中*所示的子树。

利用这个，我们就可以实现线段树的一些功能，比如回答对区间的询问。我们在每个结点上记录关于以这个结点为根的子树的信息，然后询问时先提取区间，再直接读取子树的相关信息。还可以对区间进行整体修改，这也要用到和线段树类似的延迟标记技术，就是对于每个结点，再额外记录一个或多个标记，表示以这个结点为根的子树是否被进行了某种操作，并且这种操作影响其子结点的信息值。当然，既然记录了标记，那么旋转和其他一些操作中也就要相应地将标记向下传递。

到目前为止，伸展树只是实现了线段树能够实现的功能，下面两个功能将是线段树无法办到的。

如果我们要在 a 后面插入一些数，那么我们先把这些插入的数建成一棵伸展树，我们可以利用分治法建立一棵完全平衡的二叉树，就是说每次把最中间的作为当前区间的根，然后左右递归处理，返回的时候进行维护。接着将 a 转到根，将 a 后面一个数对应的结点转到根结点的右边，最后将这棵新的子树挂到根右子结点的左子结点上。

还有一个操作就是删除一个区间 $[a,b]$ 内的数，像上面一样，我们先提取区间，然后直接删除那棵子树，即可达到目的。

最后还需注意的就是，每当进行一个对数列进行修改的操作后，都要维护伸展树，一种方法就是对影响到的结点从下往上执行 *Update* 操作。但还有一种方法，就是将修改的结点旋转到根，因为 *Splay* 操作在旋转的同时也会维护每个结点的值，因此可以达到对整个伸展树维护的目的。

最后还有一个小问题，因为数列中第一个数前面没有数字了，并且最后一个数后面也没有数字了，这样提取区间时就会出一些问题。为了不进行过多的特殊判断，我们在原数列最前面和最后面分别加上一个数，在伸展树中就体现为结点，这样提取区间的时候原来的第 k 个数就是现在的第 $k+1$ 个数。并且我们还要注意，这两个结点维护的信息不能影响到正确的结果。

下面看一下新的 *Splay* 操作的程序（能对结点信息进行维护）：

```
// node 为结点类型，其中 ch[0] 表示左结点指针，ch[1] 表示右结点指针
// pre 表示指向父亲的指针

void Rotate(node *x, int c)  // 旋转操作，c=0 表示左旋，c=1 表示右旋
  {
    node *y = x->pre;
    Push_Down(y), Push_Down(x);
    // 先将 Y 结点的标记向下传递（因为 Y 在上面），再把 X 的标记向下传递
    y->ch[! c] = x->ch[c];
    if (x->ch[c] != Null) x->ch[c]->pre = y;
    x->pre = y->pre;
    if (y->pre != Null)
      if (y->pre->ch[0] == y) y->pre->ch[0] = x; else y->pre->ch[1] = x;
    x->ch[c] = y, y->pre = x, Update(y);  // 维护 Y 结点
```

```
    if (y == root) root = x;  // root 表示整棵树的根结点
  }

void Splay(node *x, node *f)  // Splay 操作，表示把结点 x 转到结点 f 的下面
  {
    for (Push_Down(x) ; x->pre != f; )  // 一开始就将 x 的标记下传
      if (x->pre->pre == f)  // 父结点的父亲即为 f，执行单旋转
        if (x->pre->ch[0] == x) Rotate(x, 1); else Rotate(x, 0);
      else
        {
          node *y = x->pre, *z = y->pre;
          if (z->ch[0] == y)
            if (y->ch[0] == x)
              Rotate(y, 1), Rotate(x, 1);  // 一字形旋转
            else
              Rotate(x, 0), Rotate(x, 1);  // 之字形旋转
          else
            if (y->ch[1] == x)
              Rotate(y, 0), Rotate(x, 0);  // 一字形旋转
            else
              Rotate(x, 1), Rotate(x, 0);  // 之字形旋转
        }
    Update(x);  // 最后再维护 x 结点
  }
```

可能有人会问，为什么在旋转的时候只对 X 结点的父亲进行维护，而不对 X 结点进行维护，但是 *Splay* 操作的最后却又维护了 X 结点？原因很简单。因为除了一字形旋转，在 *Splay* 操作里我们进行的旋转都只对 X 结点进行，因此过早地维护是多余的；而在一字形旋转中，好像在旋转中没有对 X 的父亲进行维护，但后面紧接着就是旋转 X 结点，又会对 X 的父亲进行维护，也是没问题的。这样可以节省不少冗余的 *Update* 操作，能减小程序隐含的常数。

最后我们看看怎么样实现把数列中第 k 个数对应的结点转到想要的位置。对于这个操作，我们要记录每个以结点为根子树的大小，即包含结点的个数，然后从根开始，每次决定是向左走，还是向右走，具体见下面的代码：

```
// 找到处在中序遍历第 k 个结点，并将其旋转到结点 f 的下面
void Select(int k, node *f)
  {
    int tmp;
    node *t;
```

```
    for (t = root; ; )  // 从根结点开始
      {
        Push_Down(t);  // 由于要访问 t 的子结点，将标记下传
        tmp = t->ch[0]->size;  // 得到 t 左子树的大小
        if (k == tmp + 1) break;  // 得出 t 即为查找结点，退出循环
        if (k <= tmp)  // 第 k 个结点在 t 左边，向左走
          t = t->ch[0];
        else  // 否则在右边，而且在右子树中，这个结点不再是第 k 个
          k -= tmp + 1, t = t->ch[1];
      }
    Splay(t, f);  // 执行旋转
  }
```

（3）实例分析——NOI 2005 维护数列（Sequence）

题目的意思很简单：维护一个数列，支持以下几种操作：

1. 插入：在当前数列第 *posi* 个数字后面插入 *tot* 个数字；若在数列首位插入，则 *posi* 为 0。

2. 删除：从当前数列第 *posi* 个数字开始连续删除 *tot* 个数字。

3. 修改：从当前数列第 *posi* 个数字开始连续 *tot* 个数字统一修改为 *c* 。

4. 翻转：取出从当前数列第 *posi* 个数字开始的 *tot* 个数字，翻转后放入原来的位置。

5. 求和：计算从当前数列第 *posi* 个数字开始连续 *tot* 个数字的和并输出。

6. 求和最大子序列：求出当前数列中和最大的一段子序列，并输出最大和。

首先，对于 1、2 两个操作，前面已经分析过了。而 3、4 两个操作为修改操作，因此我们只需增加两个标记：*same* 和 *rev*，分别表示这棵子树是否被置为一个数和是否被翻转。

对于第 5 个操作，我们给每个结点维护一个 *sum*，即可解决问题。第 6 个操作的实现是最为复杂的，我们需要维护 4 个值：*value* 、*MaxL* 、*MaxR* 和 *MaxM* ，分别表示这个结点的值、这个子树表示数列左起最大连续和、右起最大连续和以及这个数列和最大的子序列。其中，*MaxL* 、*MaxR* 和 *MaxM* 的值可以由 *value* 和这个结点子树子树的相关信息得到。以 *MaxL* 为例，我们要考虑三种情况：只在左子树对应的序列中、全部左子树的序列和这个结点以及横跨左右子树（左子树

全部包含）。*MaxR* 和 *MaxM* 的维护可以类比一下（注意 *MaxM* 有五种情况）。

然后由于要执行 *Select* 操作，因此每个结点还要维护一个 *size* 。最终得到每个结点要维护 8 个标记或信息：*same* 、*rev* 、*value* 、*sum* 、*size* 、*MaxL* 、*MaxR* 、*MaxM* 。

因为我们要询问的是子序列和以及最大和子序列，因此上文中提及的额外增加的两个结点的 *value* 、*MaxL* 、*MaxR* 、*MaxM* 应为一个很小的数（比如-100000），而 *sum* 应该为 0，这样才不会影响正常的询问结果。

最后注意的是，每种修改操作（插入、删除、修改和翻转）过后，要对伸展树的信息重新维护。按照前面说的，我们将修改的结点（即根结点右子结点的左子结点） *Splay* 到根的位置。这样就能保证新的伸展树的值都是正确的。

还有一点就是删除操作要回收空间，不然会使用过多的空间（大概要 100*MB* ）。并且由于 *C++* 的指针回收操作过慢，因此我们人工压一个栈回收结点指针。

（4）和线段树的比较

用伸展树解决数列维护问题，可以支持两个线段树无法支持的操作：在某个位置插入一些数和删除一些连续的数。但是也带了更大的常数和更大的代码量。因此，在没有必要使用伸展树的时候，我们就不应该使用。不过有些问题看似线段树无法解决，然而对模型进行转化后，也能用线段树解决，所以做题的时候不要急着出算法，还是要分析一下问题的本质，选择最好最合适的方法解决。