伸展树的基本操作与应用

安徽省芜湖市第一中学 杨思雨

目录

【关键字】

伸展树 基本操作 应用

【摘要】

本文主要介绍了伸展树的基本操作以及其在解题中的应用。全文可以分为以下四个部分。

第一部分引言，主要说明了二叉查找树在信息学竞赛中的重要地位，并且指出二叉查找树在某些情况下时间复杂度较高，进而引出了在时间复杂度上更为优秀的伸展树。

第二部分介绍了伸展树的基本操作。并给出了对伸展树时间复杂度的分析和证明，指出伸展树的各种基本操作的平摊复杂度均为O(log n)，说明伸展树是一种较平衡的二叉查找树。

第三部分通过一个例子介绍了伸展树在解题中的应用，并将它与其它树状数据结构进行了对比。

第四部分指出了伸展树的优点，总结全文。

【引言】

二叉查找树（Binary Search Tree）能够支持多种动态集合操作。因此，在信息学竞赛中，二叉排序树起着非常重要的作用，它可以被用来表示有序集合、建立索引或优先队列等。

作用于二叉查找树上的基本操作的时间是与树的高度成正比的。对一个含n个节点的完全二叉树，这些操作的最坏情况运行时间为O(log n)。但如果树是含n个节点的线性链，则这些操作的最坏情况运行时间为O(n)。而有些二叉查找树的变形，其基本操作在最坏情况下性能依然很好，比如红黑树、AVL树等等。

本文将要介绍的伸展树（Splay Tree），也是对二叉查找树的一种改进，虽然它并不能保证树一直是“平衡”的，但对于伸展树的一系列操作，我们可以证明其每一步操作的平摊复杂度都是O(log n)。所以从某种意义上说，伸展树也是一种平衡的二叉查找树。而在各种树状数据结构中，伸展树的空间要求与编程复杂度也都是很优秀的。

【伸展树的基本操作】

伸展树是二叉查找树的一种改进，与二叉查找树一样，伸展树也具有有序性。即伸展树中的每一个节点x都满足：该节点左子树中的每一个元素都小于x，而其右子树中的每一个元素都大于x。与普通二叉查找树不同的是，伸展树可以自我调整，这就要依靠伸展操作Splay(x,S)。

伸展操作Splay(x,S)

伸展操作Splay(x,S)是在保持伸展树有序性的前提下，通过一系列旋转将伸展树S中的元素x调整至树的根部。在调整的过程中，要分以下三种情况分别处理：

情况一：节点x的父节点y是根节点。这时，如果x是y的左孩子，我们进行一次Zig（右旋）操作；如果x是y的右孩子，则我们进行一次Zag（左旋）操作。经过旋转，x成为二叉查找树S的根节点，调整结束。如图1所示

图 1

情况二：节点x的父节点y不是根节点，y的父节点为z，且x与y同时是各自父节点的左孩子或者同时是各自父节点的右孩子。这时，我们进行一次Zig-Zig操作或者Zag-Zag操作。如图2所示

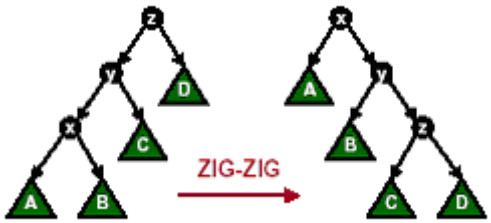

图 2

情况三：节点x的父节点y不是根节点，y的父节点为z，x与y中一个是其父节点的左孩子而另一个是其父节点的右孩子。这时，我们进行一次Zig-Zag操作或者Zag-Zig操作。如图3所示

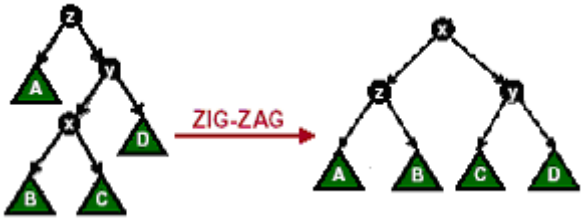

图 3

如图4所示，执行Splay(1,S)，我们将元素1调整到了伸展树S的根部。再执行Splay(2,S)，如图5所示，我们从直观上可以看出在经过调整后，伸展树比原来“平衡”了许多。而伸展操作的过程并不复杂，只需要根据情况进行旋转就可以了，而三种旋转都是由基本得左旋和右旋组成的，实现较为简单。

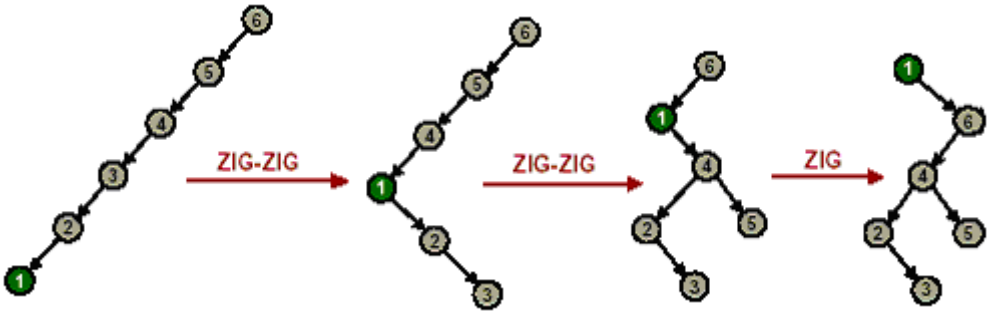

图 4 Splay(1,S)

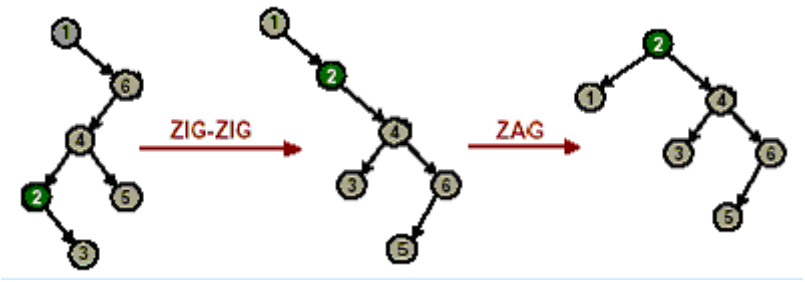

图 5 Splay(2,S)

伸展树的基本操作

利用Splay操作，我们可以在伸展树S上进行如下运算：

(1)Find(x,S)：判断元素x是否在伸展树S表示的有序集中。

首先，与在二叉查找树中的查找操作一样，在伸展树中查找元素x。如果x在树中，则再执行Splay(x,S)调整伸展树。

(2)Insert(x,S)：将元素x插入伸展树S表示的有序集中。

首先，也与处理普通的二叉查找树一样，将x插入到伸展树S中的相应位置上，再执行Splay(x,S)。

(3)Delete(x,S)：将元素x从伸展树S所表示的有序集中删除。

首先，用在二叉查找树中查找元素的方法找到x的位置。如果x没有孩子或只有一个孩子，那么直接将x删去，并通过Splay操作，将x节点的父节点调整到伸展树的根节点处。否则，则向下查找x的后继y，用y替代x的位置，最后执行Splay(y,S)，将y调整为伸展树的根。

(4)Join(S1,S2)：将两个伸展树S1与S2合并成为一个伸展树。其中S1的所有元素都小于S2的所有元素。

首先，我们找到伸展树S1中最大的一个元素x，再通过Splay(x,S1)将x调整到伸展树S1的根。然后再将S2作为x节点的右子树。这样，就得到了新的伸展

树S。如图6所示

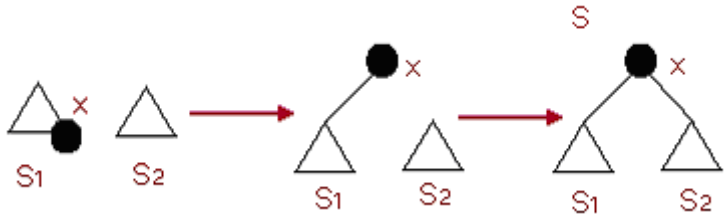

图 6

(5)Split(x,S)：以x为界，将伸展树S分离为两棵伸展树S1和S2，其中S1中所有元素都小于x，S2中的所有元素都大于x。

首先执行Find(x,S)，将元素x调整为伸展树的根节点，则x的左子树就是S1，而右子树为S2。如图7所示

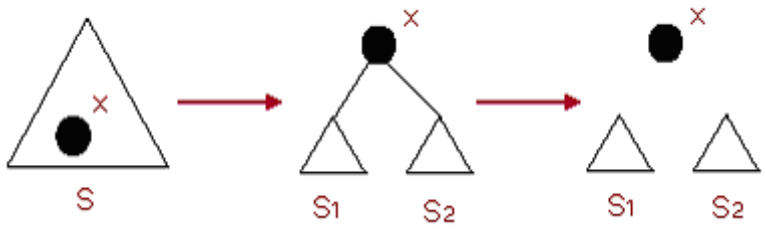

图 7

除了上面介绍的五种基本操作，伸展树还支持求最大值、求最小值、求前趋、求后继等多种操作，这些基本操作也都是建立在伸展操作的基础上的。

时间复杂度分析

由以上这些操作的实现过程可以看出，它们的时间效率完全取决于Splay操作的时间复杂度。下面，我们就用会计方法来分析Splay操作的平摊复杂度。

首先，我们定义一些符号：S(x)表示以节点x为根的子树。|S|表示伸展树S的节点个数。令 μ (S) = [log|S|]， μ (x)= μ (S(x))。如图8所示

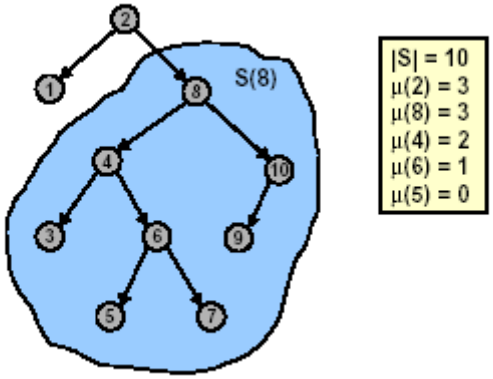

图 8

我们用1元钱表示单位代价（这里我们将对于某个点访问和旋转看作一个单位时间的代价）。定义**伸展树不变量**：在任意时刻，伸展树中的任意节点x都至少有μ(x)元的存款。

在Splay调整过程中，费用将会用在以下两个方面：

(1)为使用的时间付费。也就是每一次单位时间的操作，我们要支付1元钱。

(2)当伸展树的形状调整时，我们需要加入一些钱或者重新分配原来树中每个节点的存款，以保持不变量继续成立。

下面我们给出关于Splay操作花费的定理：

定理：在每一次Splay(x,S)操作中，调整树的结构与保持伸展树不变量的总花费不超过3μ(S)+1。

证明：用μ(x)和μ'(x)分别表示在进行一次Zig、Zig-Zig或Zig-Zag操作前后节点x处的存款。

下面我们分三种情况分析旋转操作的花费：

情况一：如图9所示

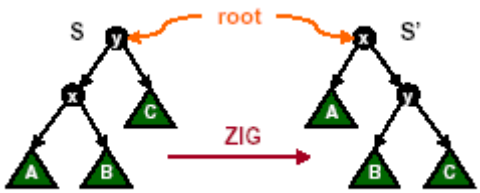

图 9

我们进行Zig或者Zag操作时，为了保持伸展树不变量继续成立，我们需要花费：

$$\begin{aligned}\mu'(x)+\mu'(y)-\mu(x)-\mu(y) &= \mu'(y)-\mu(x)\\ &\le \mu'(x)-\mu(x)\\ &\le 3(\mu'(x)-\mu(x))\\ &= 3(\mu(S)-\mu(x))\end{aligned}$$

此外我们花费另外1元钱用来支付访问、旋转的基本操作。因此，一次Zig或Zag操作的花费至多为$3(\mu(S)-\mu(x))$。

情况二：如图10所示

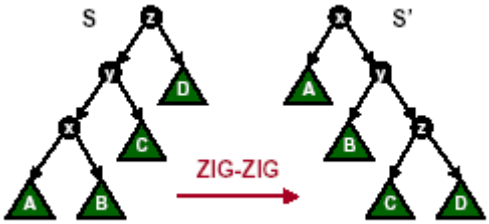

图 10

我们进行Zig-Zig操作时，为了保持伸展树不变量，我们需要花费：

$$\begin{aligned}\mu'(x)+\mu'(y)+\mu'(z)-\mu(x)-\mu(y)-\mu(z) &= \mu'(y)+\mu'(z)-\mu(x)-\mu(y)\\ &= (\mu'(y)-\mu(x))+(\mu'(z)-\mu(y))\\ &\le (\mu'(x)-\mu(x))+(\mu'(x)-\mu(x))\\ &= 2(\mu'(x)-\mu(x))\end{aligned}$$

与上种情况一样，我们也需要花费另外的1元钱来支付单位时间的操作。

当$\mu'(x)<\mu(x)$时，显然$2(\mu'(x)-\mu(x))+1 \le 3(\mu'(x)-\mu(x))$。也就是进行Zig-Zig操作的花费不超过$3(\mu'(x)-\mu(x))$。

当$\mu'(x)=\mu(x)$时，我们可以证明$\mu'(x)+\mu'(y)+\mu'(z)<\mu(x)+\mu(y)$

+μ(z)，也就是说我们不需要任何花费保持伸展树不变量，并且可以得到退回来的钱，用其中的1元支付访问、旋转等操作的费用。为了证明这一点，我们假设μ'(x)+μ'(y)+μ'(z)>μ(x)+μ(y)+μ(z)。

联系图9，我们有μ(x) =μ'(x) =μ(z)。那么，显然μ(x) =μ(y) =μ(z)。于是，可以得出μ(x) =μ'(z) =μ(z)。令a = 1 + |A| + |B|，b = 1 + |C| + |D|，那么就有

[log a] = [log b] = [log (a+b+1)]。 ①

我们不妨设b≥a，则有

[log (a+b+1)] ≥ [log (2a)]
= 1+[log a]
> [log a] ②

①与②矛盾，所以我们可以得到μ'(x) =μ(x) 时，Zig-Zig操作不需要任何花费，显然也不超过3(μ'(x) - μ(x))。

情况三：与情况二类似，我们可以证明，每次Zig-Zag操作的花费也不超过3(μ'(x) - μ(x))。

以上三种情况说明，Zig操作花费最多为3(μ(S)- μ(x))+1，Zig-Zig或Zig-Zag操作最多花费3(μ'(x)- μ(x))。那么将旋转操作的花费依次累加，则一次Splay(x,S)操作的费用就不会超过3μ(S)+1。也就是说对于伸展树的各种以Splay操作为基础的基本操作的平摊复杂度，都是O(log n)。所以说，伸展树是一种时间效率非常优秀的数据结构。

【伸展树的应用】

伸展树作为一种时间效率很高、空间要求不大的数据结构，在解题中有很大的用武之地。下面就通过一个例子说明伸展树在解题中的应用。

例：营业额统计Turnover （湖南省队2002年选拔赛）

题目大意

Tiger最近被公司升任为营业部经理，他上任后接受公司交给的第一项任务便是统计并分析公司成立以来的营业情况。Tiger拿出了公司的账本，账本上记录了公司成立以来每天的营业额。分析营业情况是一项相当复杂的工作。由于节假日，大减价或者是其他情况的时候，营业额会出现一定的波动，当然一定的波动是能够接受的，但是在某些时候营业额突变得很高或是很低，这就证明公司此时的经营状况出现了问题。经济管理学上定义了一种**最小波动值**来衡量这种情况：

该天的最小波动值= min { | 该天以前某一天的营业额-该天的营业额 | }

当最小波动值越大时，就说明营业情况越不稳定。而分析整个公司的从成立到现在营业情况是否稳定，只需要把每一天的最小波动值加起来就可以了。你的任务就是编写一个程序帮助Tiger来计算这一个值。

注：第一天的最小波动值为第一天的营业额。

数据范围：天数n≤32767，每天的营业额ai≤1,000,000。最后结果T≤2^{31}。

初步分析

题目的意思非常明确，关键是要每次读入一个数，并且在前面输入的数中找到一个与该数相差最小的一个。

我们很容易想到$O(n^2)$的算法：每次读入一个数，再将前面输入的数一次查找一遍，求出与当前数的最小差值，记入总结果T。但由于本题中n很大，这样的算法是不可能在时限内出解的。而如果使用线段树记录已经读入的数，就需要记下一个2M的大数组，这在当时比赛使用TurboPascal 7.0编程的情况下是不可能实现的。而前文提到的红黑树与平衡二叉树虽然在时间效率、空间复杂度上都比较优秀，但过高的编程复杂度却让人望而却步。于是我们想到了伸展树算法。

算法描述

进一步分析本题，解题中，涉及到对于有序集的三种操作：插入、求前趋、求后继。而对于这三种操作，伸展树的时间复杂度都非常优秀，于是我们设计了如下算法：

开始时，树S为空，总和T为零。每次读入一个数p，执行Insert(p,S)，将p插入伸展树S。这时，p也被调整到伸展树的根节点。这时，求出p点左子树中的最

右点和右子树中的最左点，这两个点分别是有序集中p的前趋和后继。然后求得最小差值，加入最后结果T。

解题小结

由于对于伸展树的基本操作的平摊复杂度都是O(log n)的，所以整个算法的时间复杂度是O(nlog n)，可以在时限内出解。而空间上，可以用数组模拟指针存储树状结构，这样所用内存不超过400K，在TP中使用动态内存就可以了。编程复杂度方面，伸展树算法非常简单，程序并不复杂。虽然伸展树算法并不是本题唯一的算法，但它与其他常用的数据结构相比还是有很多优势的。下面的表格就反映了在解决这一题时各个算法的复杂度。从中可以看出伸展树在各方面都是优秀的，这样的算法很适合在竞赛中使用。

	顺序查找	线段树	AVL树	伸展树
时间复杂度	$O(n^2)$	O(nlog a)	O(nlog n)	O(nlog n)
空间复杂度	O(n)	O(a)	O(n)	O(n)
编程复杂度	很简单	较简单	较复杂	较简单

【总结】

由上面的分析介绍，我们可以发现伸展树有以下几个优点：

(1)时间复杂度低，伸展树的各种基本操作的平摊复杂度都是O(log n)的。在树状数据结构中，无疑是非常优秀的。

(2)空间要求不高。与红黑树需要记录每个节点的颜色、AVL树需要记录平衡因子不同，伸展树不需要记录任何信息以保持树的平衡。

(3)算法简单，编程容易。伸展树的基本操作都是以Splay操作为基础的，而Splay操作中只需根据当前节点的位置进行旋转操作即可。

虽然伸展树算法与AVL树在时间复杂度上相差不多，甚至有时候会比AVL树慢一些，但伸展树的编程复杂度大大低于AVL树。在竞赛中，使用伸展树在编程和调试中都更有优势。

在信息学竞赛中，不能只一味的追求算法有很高的时间效率，而需要在时

间复杂度、空间复杂度、编程复杂度三者之间找到一个“平衡点”，合理的选择算法。这也需要我们在平时对各种算法反复琢磨，深入研究，在竞赛中才能够游刃有余的应用。

【参考书目】

[1]傅清祥，王晓东.《算法与数据结构》.电子工业出版社.1998.01
[2]严蔚敏，吴伟民.《数据结构(第二版)》.清华大学出版社.1992.06
[3]Thomas H.Cormen，Charles E.Leiserson，Ronald L.Rivest，Clifford Stein《Introduction to Algorithms,Second Edition》.The MIT Press.2001

【附录】

(一)伸展操作和其他各种基本操作，其实现参见：Splay Tree.doc

1伸展树类型的定义

```
Type
Tpoint=^Trec;
Trec=Record
        Key:integer;                                    //关键字
        Father,LeftChild,RightChild:Tpoint;
//父亲、左孩子、右孩子
      End;
 Var
      S:Tpoint;
//变量S记录根节点
```

2伸展操作

2.1左旋操作

```
Procedure LeftRotate(x:Tpoint);
Var y:Tpoint;
Begin
    y:=x^.Father;

    y^.RightChild:=x^.LeftChild;
    if x^.LeftChild<>nil
       then x^.LeftChild^.Father:=y;

    x^.Father:=y^.Father;
    if y^.Father<>nil then begin
        if y=y^.Father^.LeftChild
            then y^.Father^.LeftChild:=x
```

```
                else y^.Father^.RightChild:=x;
          end;
        y^.Father:=x;
        x^.LeftChild:=y;
    End;
```

2.2右旋操作

```
Procedure RightRotate(x:Tpoint);
Var y:Tpoint;
Begin
    y:=x^.Father;

    y^.LeftChild:=x^.RightChild;
    if x^.RightChild<>nil
       then x^.RightChild^.Father:=y;

    x^.Father:=y^.Father;
    if y^.Father<>nil then begin
       if y=y^.Father^.LeftChild
                then y^.Father^.LeftChild:=x
                else y^.Father^.RightChild:=x;
    end;
    y^.Father:=x;
    x^.RightChild:=y;
End;
```

2.3伸展操作

```
Procedure Splay(var x,S:Tpoint);
Var p:Tpoint;
Begin
    while x^.Father<>nil do
          begin
                p:=x^.Father;

              if p^.Father=nil then begin
                   if x=p^.LeftChild
                 then                        RightRotate(x)
//Zig
                   else                      LeftRotate(x);
//Zag
                   break;
               end;

                if x=p^.LeftChild then begin
```

```
                        if p=p^.Father^.LeftChild then begin
                          RightRotate(p);
    //Zig-Zig
                             RightRotate(x);
                         end
                          else begin
                             RightRotate(x);
    //Zig-Zag
                             LeftRotate(x);
                          end;
                       end
                       else begin
                          if p=p^.Father^.RightChild then begin
                          LeftRotate(p);
    //Zag-Zag
                            LeftRotate(x);
                        end
                          else begin
                              LeftRotate(x);
    //Zag-Zig
                              RightRotate(x);
                          end;
                     end;
                end;

            S:=x;
    //x为树的根节点
        End;
```

3查找

```
Function Find(x:integer; var S:Tpoint):Tpoint;
Var p:Tpoint;
Begin
     p:=BST_Search(x,S);
```

//与普通二叉查找树相同

找到该节点

```
        Splay(p,S);
```

//将该节点调整到根部

```
        Find:=p;
End;
```

4插入

```
Procedure Insert(var x,S:Tpoint);
```

```
        Begin
            BST_Insert(x,S);
//与普通二叉查找树相同
                                                        插入该元素
            Splay(x,S);
//将该节点调整到根部
        End;
```

5删除

```
        Procedure Delete(x:integer; var S:Tpoint);
        Var p:Tpoint;
        Begin
                p:=Find(x,S);
//找到这个节点，x成为树根
                S:=Join(p^.LeftChild,p^.RightChild);
                                                    //合并操作见下文“10
合并”
        End;
```

6求最大值

```
        Function Maximum(S:Tpoint):Tpoint;
        Var p:Tpoint;
        Begin
            p:=S;
            while p^.RightChild<>nil do p:=p^.RightChild;
                                                        //树最右下端就是最
大值
            Splay(p,S);
//将最大值调整为根
            Maximum:=p;
        End;
```

7求最小值

```
        Function Minimum(S:Tpoint):Tpoint;
        Var p:Tpoint;
        Begin
            p:=S;
            while p^.LeftChild<>nil do p:=p^.LeftChild;
                                                        //树最左下端就是最
小值
            Splay(p,S);
//将最小值调整为根
            Minimum:=p;
        End;
```

8求前趋

```
Function Predecessor(x:integer; var S:Tpoint);
Var p:Tpoint;
Begin
    p:=Find(x,S);                                  //将x调整为根
    p:=p^.LeftChild;
    Predecessor:=Maximum(p);
//前趋即为左子树的最大值
End;
```

9求后继

```
Function Successor(x:integer; var S:Tpoint);
Var p:Tpoint;
Begin
    p:=Find(x,S);                                  //将x调整为根
    p:=p^.RightChild;
    Successor:=Minimum(p);
//后继即为右子树的最小值
End;
```

10合并

```
Function Join(S1,S2:Tpoint):Tpoint;
Var p:Tpoint;
Begin
    if S1=nil then begin
        Join:=S2; exit;
    end;
    if S2=nil then begin
        Join:=S1; exit;
    end;

    p:=Maximum(S1);
//找到S1中的最大值p
    p^.RightChild:=S2;
//S2成为p的右孩子
    Join:=p;
End;
```

11分离

```
Procedure Split(x:integer; var S,S1,S2:Tpoint);
Var p:Tpoint;
Begin
    p:=Find(x,S);
```

```
//将x调整为根
            S1:=p^.LeftChild;
//左孩子为S1
            S2:=p^.RightChild;
//右孩子为S2
        End;
```

(二)文中提到的伸展树的基本操作，具体过程可参照动画：Splay Tree.htm

(三)针对文中例题，作者用伸展树算法编写了程序：Turnover.pas

```
Program Turnover_SplayTree_FP;
Const
    inf='turnover.in';
    outf='turnover.out';
    maxn=32767;

Type
    Tp=record                                   //树的节点
           data:longint;                        //营业额
           father,left,right:integer;
       end;

Var
    n,now,
//总天数、当前天数
    root,                                       //伸展树根
    prev,next,                                  //前趋、后继
    same:integer;
//当天营业额相同的
    min:longint;                                //最小差值
    ans:extended;                               //答案
    tree:array[0..maxn]of Tp;

procedure                                       init;
//初始化树，输入
begin
    fillchar(tree,sizeof(tree),0);
    read(n);                                    //输入天数
    read(tree[1].data);
    root:=1;
    ans:=tree[1].data;
end;
```

```
procedure leftrotate(x:integer);              //左旋
var y:integer;
begin
    y:=tree[x].father;

    tree[y].right:=tree[x].left;
    if tree[x].left<>0
       then tree[tree[x].left].father:=y;

    tree[x].father:=tree[y].father;
    if tree[y].father<>0 then begin
        if y=tree[tree[y].father].left
          then tree[tree[y].father].left:=x
          else tree[tree[y].father].right:=x;
    end;
    tree[y].father:=x;
    tree[x].left:=y;
end;

procedure rightrotate(x:integer);             //右旋
var y:integer;
begin
    y:=tree[x].father;

    tree[y].left:=tree[x].right;
    if tree[x].right<>0
         then tree[tree[x].right].father:=y;

    tree[x].father:=tree[y].father;
    if tree[y].father<>0 then begin
        if y=tree[tree[y].father].left
          then tree[tree[y].father].left:=x
          else tree[tree[y].father].right:=x;
    end;

    tree[y].father:=x;
    tree[x].right:=y;
end;

procedure splay(now:integer);                 //伸展操作
var t:integer;
begin
    while tree[now].father<>0 do
```

```
      begin
          t:=tree[now].father;

          if tree[t].father=0 then begin
              if now=tree[t].left
{ZIG}           then rightrotate(now)
{ZAG}           else leftrotate(now);
              break;
          end;

          if now=tree[t].left then begin
              if t=tree[tree[t].father].left then begin
{ZIG-ZIG}         rightrotate(t);
                  rightrotate(now);
              end
              else begin
{ZIG-ZAG}         rightrotate(now);
                  leftrotate(now);
              end;
          end
          else begin
              if t=tree[tree[t].father].right then begin
{ZAG-ZAG}         leftrotate(t);
                  leftrotate(now);
              end
              else begin
{ZAG-ZIG}         leftrotate(now);
                  rightrotate(now);
              end;
          end;
      end;

    root:=now;
end;

procedure insert;                                    //插入
var t:integer;
begin
    t:=root; same:=now;
    while true do
      begin
          if tree[t].data=tree[now].data
            then begin min:=0; same:=t; exit; end;
          if tree[t].data>tree[now].data then begin
```

```
                if tree[t].left=0 then begin
                    tree[now].father:=t;
                    tree[t].left:=now;
                    exit;
                end
                else t:=tree[t].left;
            end
            else begin
                if tree[t].right=0 then begin
                   tree[now].father:=t;
                   tree[t].right:=now;
                   exit;
                end
                else t:=tree[t].right;
            end;
        end;
end;

procedure findprev;                                    //找前趋
begin
    prev:=tree[now].left;
    while prev<>0 do
      begin
          if tree[prev].right=0 then exit;
          prev:=tree[prev].right;
      end;
end;

procedure findnext;                                    //找后继
begin
    next:=tree[now].right;
    while next<>0 do
      begin
          if tree[next].left=0 then exit;
          next:=tree[next].left;
      end;
end;

procedure work;
begin
    read(tree[now].data);
    min:=maxlongint;

    insert;                                             //插入
```

```
            splay(same);                                          //伸展操作
            if min>0 then begin
              findprev;
//求前趋
              findnext;
//求后继
              if (next<>0) and ((prev=0) or
                 (tree[next].data-tree[now].data<
                   tree[now].data-tree[prev].data))
              then min:=tree[next].data-tree[now].data
              else min:=tree[now].data-tree[prev].data;  //求最小差值
            end;

            ans:=ans+min;
        end;

        procedure main;                                           //主过程
        begin
            assign(input,inf); reset(input);
            assign(output,outf); rewrite(output);
              init;
              for now:=2 to n do work;
              writeln(ans:0:0);
            close(input);
            close(output);
        end;

        Begin
            main;
        End.
```