左偏树的特点及其应用

广东省中山市第一中学　黄源河

【摘要】

本文较详细地介绍了左偏树的特点以及它的各种操作。

第一部分提出可并堆的概念，指出二叉堆的不足，并引出左偏树。第二部分主要介绍了左偏树的定义和性质。第三部分详细地介绍了左偏树的各种操作，并给出时间复杂度分析。第四部分通过一道例题，说明左偏树在当今信息学竞赛中的应用。第五部分对各种可并堆作了一番比较。最后总结出左偏树的特点以及应用前景。

【关键字】　左偏树　可并堆　优先队列

【目录】

【正文】

一、引言

优先队列在信息学竞赛中十分常见，在统计问题、最值问题、模拟问题和贪心问题等等类型的题目中，优先队列都有着广泛的应用。二叉堆是一种常用的优先队列，它编程简单，效率高，但如果问题需要对两个优先队列进行合并，二叉堆的效率就无法令人满意了。本文介绍的左偏树，可以很好地解决这类问题。

二、左偏树的定义和性质

在介绍左偏树之前，我们先来明确一下优先队列和可并堆的概念。

2.1 优先队列，可并堆

2.1.1 优先队列的定义

优先队列(Priority Queue)是一种抽象数据类型(ADT)，它是一种容器，里面有一些元素，这些元素也称为队列中的节点(node)。优先队列的节点至少要包含一种性质：有序性，也就是说任意两个节点可以比较大小。为了具体起见我们假设这些节点中都包含一个键值(key)，节点的大小通过比较它们的键值而定。优先队列有三个基本的操作：插入节点(Insert)，取得最小节点(Minimum) 和删除最小节点(Delete-Min)。

2.1.2 可并堆的定义

可并堆(Mergeable Heap)也是一种抽象数据类型，它除了支持优先队列的三个基本操作(Insert, Minimum, Delete-Min)，还支持一个额外的操作——合并操作：

$$H \leftarrow \mathrm{Merge}(H_1,H_2)$$

Merge( ) 构造并返回一个包含 H_1 和 H_2 所有元素的新堆 H。

前面已经说过，如果我们不需要合并操作，则二叉堆是理想的选择。可惜合并二叉堆的时间复杂度为 O(n)，用它来实现可并堆，则合并操作必然成为算法的瓶颈。左偏树(Leftist Tree)、二项堆(Binomial Heap) 和 Fibonacci 堆(Fibonacci Heap) 都是十分优秀的可并堆。本文讨论的是左偏树，在后面我们将看到各种可并堆的比较。

2.2 左偏树的定义

左偏树(Leftist Tree)是一种可并堆的实现。左偏树是一棵二叉树，它的节点除了和二叉树的节点一样具有左右子树指针(left, right) 外，还有两个属性：键值和距离(dist)。键值上面已经说过，是用于比较节点的大小。距离则是如下定义的：

节点 i 称为**外节点(external node)**，当且仅当节点 i 的左子树或右子树为空 (left(i) = NULL 或 right(i) = NULL)；节点 i 的**距离(dist(i))** 是节点 i 到它的后代中，最近的外节点所经过的边数。特别的，如果节点 i 本身是外节点，则它的距离为 0；而空节点的距离规定为-1 (dist(NULL) = -1)。在本文中，有时也提到一棵左偏树的距离，这指的是该树根节点的距离。

左偏树满足下面两条基本性质：

［性质 1］ 节点的键值小于或等于它的左右子节点的键值。

即 **key(i)≤key(parent(i))** 这条性质又叫**堆性质**。符合该性质的树是**堆有序的(Heap-Ordered)**。有了性质 1，我们可以知道左偏树的根节点是整棵树的最小节点，于是我们可以在 O(1) 的时间内完成取最小节点操作。

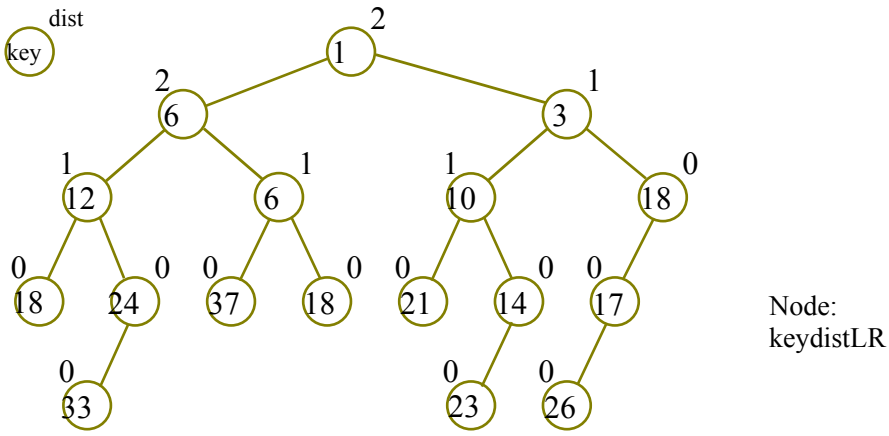
［性质 2］ 节点的左子节点的距离不小于右子节点的距离。

即 **dist(left(i))≥dist(right(i))** 这条性质称为**左偏性质**。性质 2 是为了使我们可以以更小的代价在优先队列的其它两个基本操作（插入节点、删除最小节点）进行后维持堆性质。在后面我们就会看到它的作用。

这两条性质是对每一个节点而言的，因此可以简单地从中得出，左偏树的左右子树都是左偏树。

由这两条性质，我们可以得出左偏树的定义：**左偏树是具有左偏性质的堆有序二叉树**。

下图是一棵左偏树：



2.3 左偏树的性质

在前面一节中，本文已经介绍了左偏树的两个基本性质，下面本文将介绍左偏树的另外两个性质。

我们知道，一个节点必须经由它的子节点才能到达外节点。由于性质 2，一个节点的距离实际上就是这个节点一直沿它的右边到达一个外节点所经过的边数，也就是说，我们有

［性质 3］ 节点的距离等于它的右子节点的距离加 1。

即 **dist(i) = dist(right(i)) + 1** 外节点的距离为 0，由于性质 2，它的右子节点必为空节点。为了满足性质 3，故前面规定空节点的距离为-1。

我们的印象中，平衡树是具有非常小的深度的，这也意味着到达任何一个节点所经过的边数很少。左偏树并不是为了快速访问所有的节点而设计的，它的

目的是快速访问最小节点以及在对树修改后快速的恢复堆性质。从图中我们可以看到它并不平衡，由于性质 2 的缘故，它的结构偏向左侧，不过距离的概念和树的深度并不同，左偏树并不意味着左子树的节点数或是深度一定大于右子树。

下面我们来讨论左偏树的距离和节点数的关系。

［引理 1］ 若左偏树的距离为一定值，则节点数最少的左偏树是完全二叉树。

证明： 由性质 2 可知，当且仅当对于一棵左偏树中的每个节点 i，都有 dist(left(i)) = dist(right(i)) 时，该左偏树的节点数最少。显然具有这样性质的二叉树是完全二叉树。

［定理 1］ 若一棵左偏树的距离为 k，则这棵左偏树至少有 2^{k+1}-1 个节点。

证明： 由引理 1 可知，当这样的左偏树节点数最少的时候，是一棵完全二叉树。距离为 k 的完全二叉树高度也为 k，节点数为 2^{k+1}-1，所以距离为 k 的左偏树至少有 2^{k+1}-1 个节点。

作为定理 1 的推论，我们有：

［性质 4］ 一棵 N 个节点的左偏树距离最多为⌊log(N+1)⌋ -1。

证明： 设一棵 N 个节点的左偏树距离为 k，由定理 1 可知，$N \geq 2^{k+1}$-1，因此 $k \leq \lfloor \log(N+1) \rfloor$ -1。

有了上面的 4 个性质，我们可以开始讨论左偏树的操作了。

三、左偏树的操作

本章将讨论左偏树的各种操作，包括插入新节点、删除最小节点、合并左偏树、构建左偏树和删除任意节点。由于各种操作都离不开合并操作，因此我们先讨论合并操作。

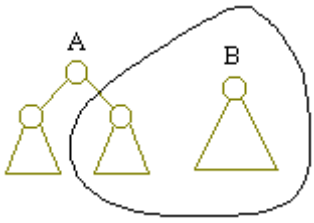
3.1 左偏树的合并

C ← Merge(A,B)

Merge( ) 把 A,B 两棵左偏树合并，返回一棵新的左偏树 C，包含 A 和 B 中的所有元素。在本文中，一棵左偏树用它的根节点的指针表示。

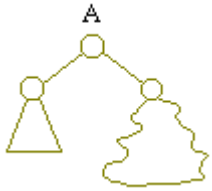
在合并操作中，最简单的情况是其中一棵树为空（也就是，该树根节点指针为 NULL）。这时我们只须要返回另一棵树。

若 A 和 B 都非空，我们假设 A 的根节点小于等于 B 的根节点（否则交换 A,B），把 A 的根节点作为新树 C 的根节点，剩下的事就是合并 A 的右子树 right(A) 和 B 了。



right(A) ← Merge(right(A), B)

合并了 right(A) 和 B 之后，right(A) 的距离可能会变大，当 right(A) 的距离大于 left(A) 的距离时，左偏树的性质 2 会被破坏。在这种情况下，我们只须要交换 left(A) 和 right(A)。



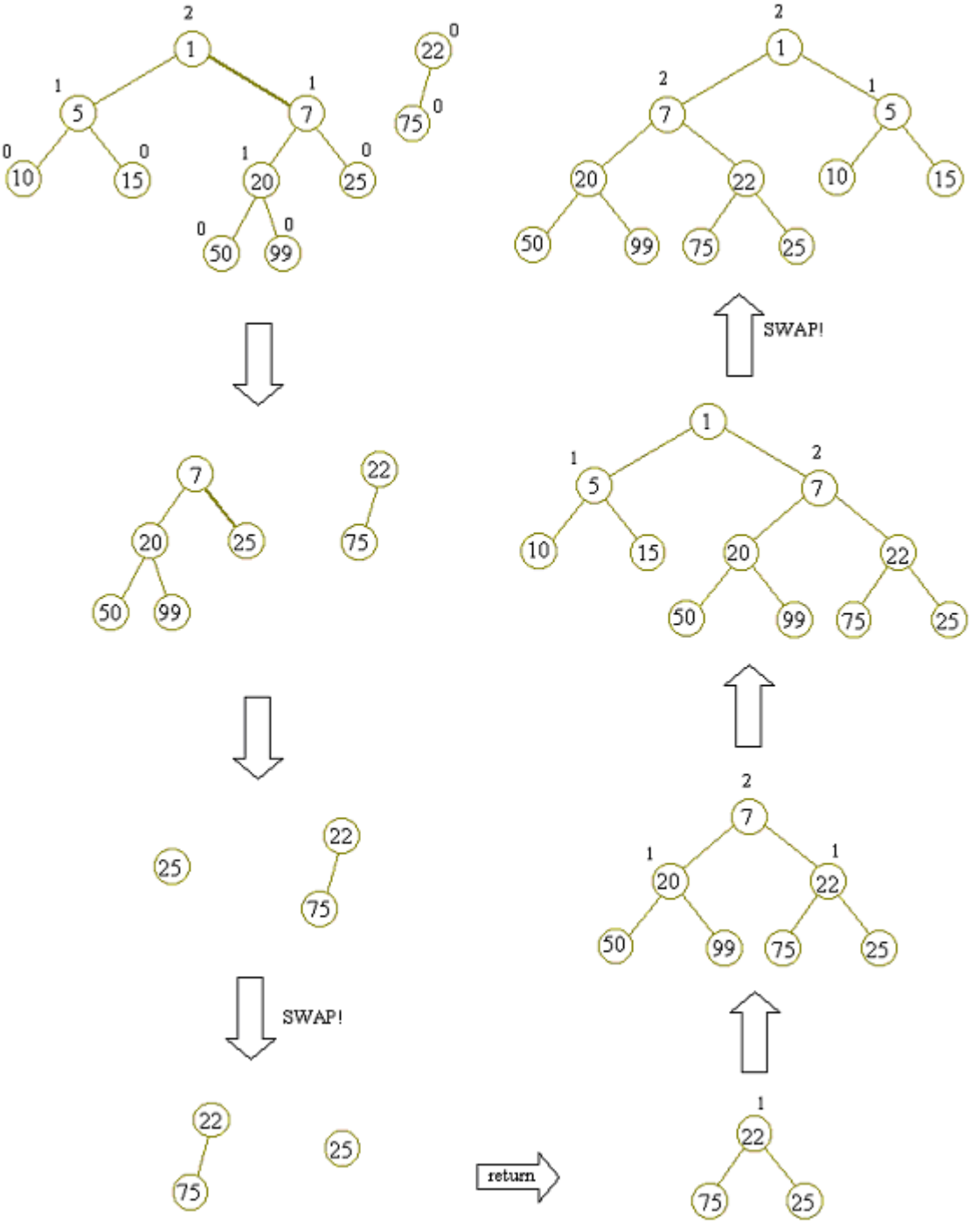
若 dist(left(A)) > dist(right(A))，交换 left(A) 和 right(A)

最后，由于 right(A) 的距离可能发生改变，我们必须更新 A 的距离：

$$dist(A) \leftarrow dist(right(A)) + 1$$

不难验证，经这样合并后的树 C 符合性质 1 和性质 2，因此是一棵左偏树。至此左偏树的合并就完成了。

下图是一个合并过程的示例：



合并流程

我们可以用下面的代码描述左偏树的合并过程：

```
Function Merge(A, B)
    If A = NULL Then return B
    If B = NULL Then return A
    If key(B) < key(A) Then swap(A, B)
    right(A) ← Merge(right(A), B)
    If dist(right(A)) > dist(left(A)) Then
        swap(left(A), right(A))
    If right(A) = NULL Then dist(A) ← 0
    Else dist(A) ← dist(right(A)) + 1
    return A
End Function
```

下面我们来分析合并操作的时间复杂度。从上面的过程可以看出，每一次递归合并的开始，都需要分解其中一棵树，总是把分解出的右子树参加下一步的合并。根据性质 3，一棵树的距离决定于其右子树的距离，而右子树的距离在每次分解中递减，因此每棵树 A 或 B 被分解的次数分别不会超过它们各自的距离。根据性质 4，分解的次数不会超过$\lfloor \log(N_1+1) \rfloor + \lfloor \log(N_2+1) \rfloor - 2$，其中 N_1 和 N_2 分别为左偏树 A 和 B 的节点个数。因此合并操作最坏情况下的时间复杂度为 $O(\lfloor \log(N_1+1) \rfloor + \lfloor \log(N_2+1) \rfloor - 2) = O(\log N_1 + \log N_2)$。

3.2 插入新节点

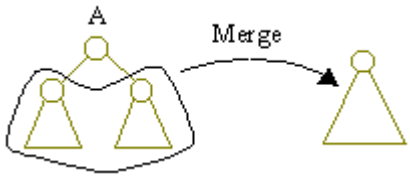
单节点的树一定是左偏树，因此向左偏树插入一个节点可以看作是对两棵左偏树的合并。下面是插入新节点的代码：

```
Procedure Insert(x, A)
    B ← MakeIntoTree(x)
    A ← Merge(A, B)
End Procedure
```

由于合并的其中一棵树只有一个节点，因此插入新节点操作的时间复杂度

是 O(log n)。

3.3 删除最小节点



由性质 1，我们知道，左偏树的根节点是最小节点。在删除根节点后，剩下的两棵子树都是左偏树，需要把他们合并。删除最小节点操作的代码也非常简单：

```
Function  DeleteMin(A)
    t ← key(root(A))
    A ← Merge(left(A), right(A))
    return t
End  Function
```

由于删除最小节点后只需进行一次合并，因此删除最小节点的时间复杂度也为 O(log n)。

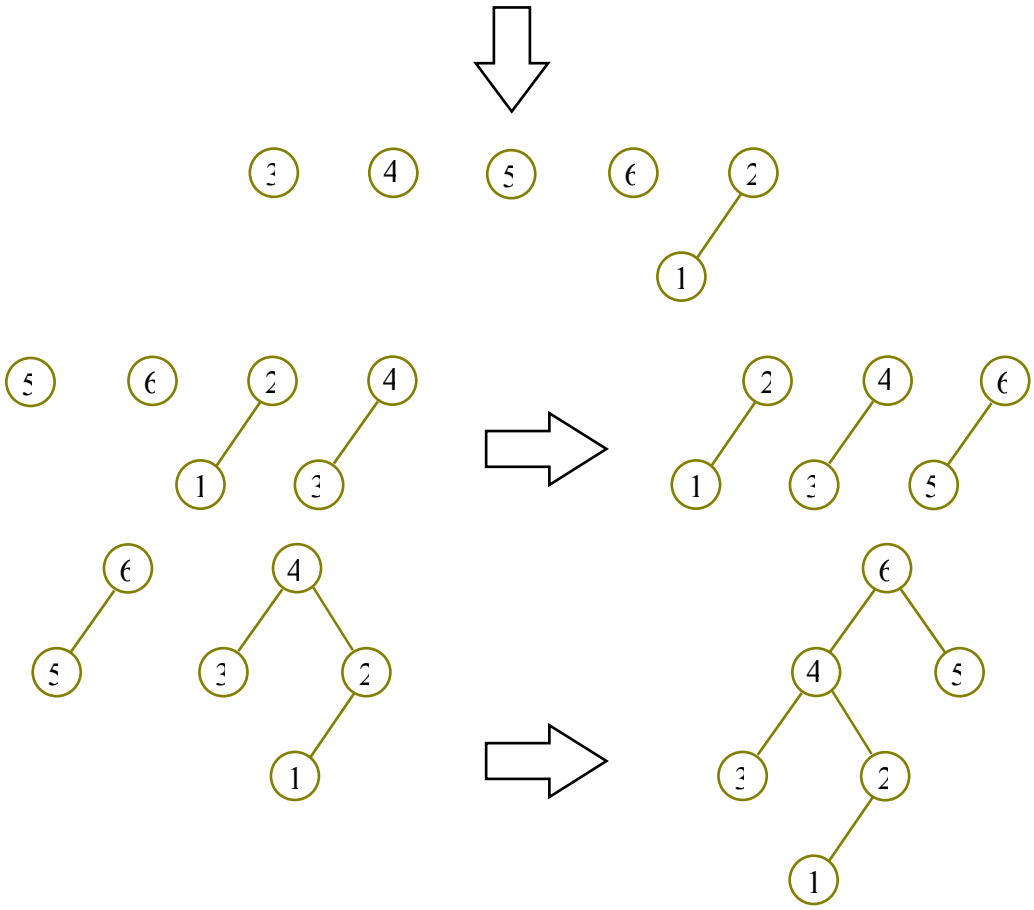
3.4 左偏树的构建

将 n 个节点构建成一棵左偏树，这也是一个常用的操作。

算法一　暴力算法——逐个节点插入，时间复杂度为 O(n log n)。

算法二　仿照二叉堆的构建算法，我们可以得到下面这种算法：

- 将 n 个节点（每个节点作为一棵左偏树）放入先进先出队列。
- 不断地从队首取出两棵左偏树，将它们合并之后加入队尾。
- 当队列中只剩下一棵左偏树时，算法结束。



构建流程

下面分析算法二的时间复杂度。假设 $n=2^k$，则：

前 $\frac{n}{2}$ 次和并的是两棵只有 1 个节点的左偏树。

接下来的 $\frac{n}{4}$ 次合并的是两棵有 2 个节点的左偏树。

接下来的 $\frac{n}{8}$ 次合并的是两棵有 4 个节点的左偏树。

……

接下来的 $\frac{n}{2^i}$ 次合并的是两棵有 2^{i-1} 个节点的左偏树。

合并两棵 2^i 个节点的左偏树时间复杂度为 O(i)，因此算法二的总时间复杂度为：$\frac{n}{2}*O(1)+\frac{n}{4}*O(2)+\frac{n}{8}*O(3)+\ldots=O(n*\sum_{i=1}^{k}\frac{i}{2^i})=O(n*(2-\frac{k+2}{2^k}))=O(n)$。

3.5 删除任意已知节点

接下来是关于删除任意已知节点的操作。之所以强调“已知”，是因为这里所说的任意节点并不是根据它的键值找出来的，左偏树本身除了可以迅速找到最小节点外，不能有效的搜索指定键值的节点。故此，我们不能要求：请删除所有键值为 100 的节点。

前面说过，优先队列是一种容器。对于通常的容器来说，一旦节点被放进去以后，容器就完全拥有了这个节点，每个容器中的节点具有唯一的对象掌握它的拥有权（ownership）。对于这种容器的应用，优先队列只能删除最小节点，因为你根本无从知道它的其它节点是什么。

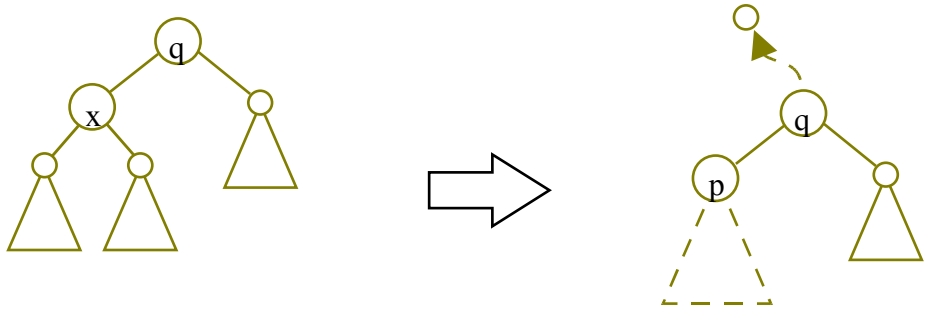
但是优先队列除了作为一种容器外还有另一个作用，就是可以找到最小节点。很多应用是针对这个功能的，它们并没有将拥有权完全转移给优先队列，而是把优先队列作为一个最小节点的选择器，从一堆节点中依次将它们选出来。这样一来节点的拥有权就可能同时被其它对象掌握。也就是说某个节点虽不是最小节点，不能从优先队列那里“已知”，但却可以从其它的拥有者那里“已知”。

这种优先队列的应用也是很常见的。设想我们有一个闹钟，它可以记录很多个响铃时间，不过由于时间是线性的，铃只能一个个按先后次序响，优先队列就很适合用来作这样的挑选。另一方面使用者应该可以随时取消一个“已知”的响铃时间，这就需要进行任意已知节点的删除操作了。

我们的这种删除操作需要指定被删除的节点，这和原来的删除根节点的操作是兼容的，因为根节点肯定是已知的。上面已经提过，在删除一个节点以后，将会剩下它的两棵子树，它们都是左偏树，我们先把它们合并成一棵新的左偏树。

$$p \leftarrow \text{Merge}(\text{left}(x), \text{right}(x))$$

现在 p 指向了这颗新的左偏树，如果我们删除的是根节点，此时任务已经完成了。不过，如果被删除节点 x 不是根节点就有点麻烦了。这时 p 指向的新树的距离有可能比原来 x 的距离要大或小，这势必有可能影响原来 x 的父节点 q 的距离，因为 q 现在成为新树 p 的父节点了。于是就要仿照合并操作里面的做法，对 q 的左右子树作出调整，并更新 q 的距离。这一过程引起了连锁反应，我们要顺着 q 的父节点链一直往上进行调整。



新树 p 的距离为 dist(p)，如果 dist(p)+1 等于 q 的原有距离 dist(q)，那么不管 p 是 q 的左子树还是右子树，我们都不需要对 q 进行任何调整，此时删除操作也就完成了。

如果 dist(p)+1 小于 q 的原有距离 dist(q)，那么 q 的距离必须调整为 dist(p)+1，而且如果 p 是左子树的话，说明 q 的左子树距离比右子树小，必须交换子树。由于 q 的距离减少了，所以 q 的父节点也要做出同样的处理。

剩下就是另外一种情况了，那就是 p 的距离增大了，使得 dist(p)+1 大于 q 的原有距离 dist(q)。在这种情况下，如果 p 是左子树，那么 q 的距离不会改变，此时删除操作也可以结束了。如果 p 是右子树，这时有两种可能：一种是 p 的距离仍小于等于 q 的左子树距离，这时我们直接调整 q 的距离就行了；另一种是 p 的距离大于 q 的左子树距离，这时我们需要交换 q 的左右子树并调整 q 的距离，交换完了以后 q 的右子树是原来的左子树，它的距离加 1 只能等于或大于 q 的原有距离，如果等于成立，删除操作可以结束了，否则 q 的距离将增大，我们还要对 q 的父节点做出相同的处理。

删除任意已知节点操作的代码如下：

```
Procedure Delete(x)
    q ← parent(x)
    p ← Merge(left(x), right(x))
    parent(p) ← q
    If q ≠ NULL and left(q) = x Then
        left(q) ← p
    If q ≠ NULL and right(q) = x Then
        right(q) ← p
    While q ≠ NULL Do
        If dist(left(q)) < dist(right(q)) Then
            swap(left(q), right(q))
        If dist(right(q))+1 = dist(q) Then
            Exit Procedure
        dist(q) ← dist(right(q))+1
        p ← q
        q ← parent(q)
    End While
End Procedure
```

下面分两种情况讨论删除操作的时间复杂度。

情况 1：p 的距离减小了。在这种情况下，由于 q 的距离只能缩小，当循环结束时，要么根节点处理完了，q 为空；要么 p 是 q 的右子树并且 dist(p)+1=dist(q)；如果 dist(p)+1>dist(q)，那么 p 一定是 q 的左子树，否则会出现 q 的右子树距离缩小了，但是加 1 以后却大于 q 的距离的情况，不符合左偏树的性质 3。不论哪种情况，删除操作都可以结束了。注意到，每一次循环，p 的距离都会加 1，而在循环体内，dist(p)+1 最终将成为某个节点的距离。根据性质 4，任何的距离都不会超过 $\log n$，所以循环体的执行次数不会超过 $\log n$。

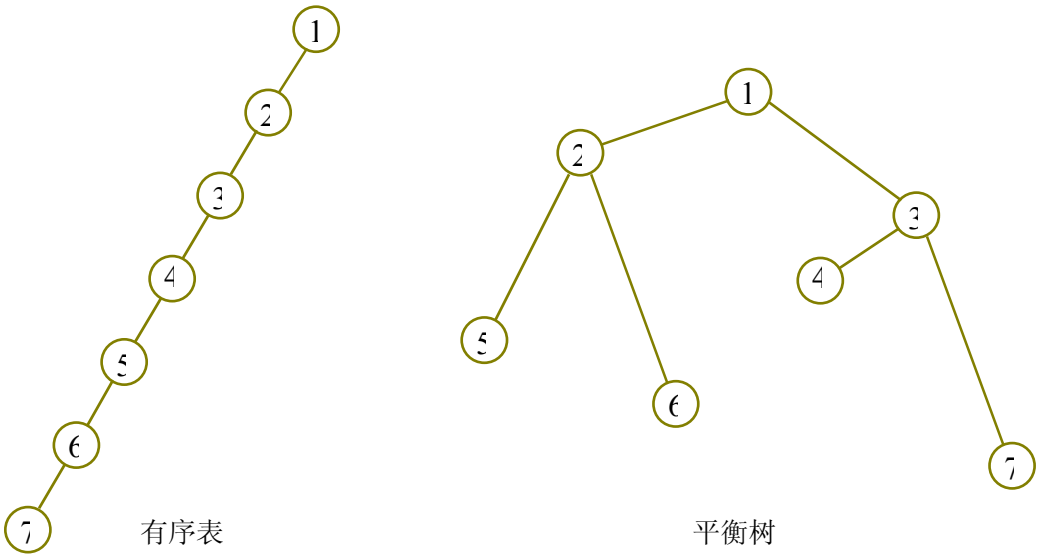
情况 2：p 的距离增大了。在这种情况下，我们将必然一直从右子树向上调整，直至 q 为空或 p 是 q 的左子树时停止。一直从右子树升上来这个事实说明了循环的次数不会超过 $\log n$（性质 4）。

最后我们看到这样一个事实，就是这两种情况只会发生其中一个。如果某种情况的调整结束后，我们已经知道要么 q 为空，要么 dist(p)+1 = dist(q)，要么 p 是 q 的左子树。这三种情况都不会导致另一情况发生。直观上来讲，如果合并后的新子树导致了父节点的一系列距离调整的话，要么就一直是往小调整，要么是一直往大调整，不会出现交替的情况。

我们已经知道合并出新子树 p 的复杂度是 O(log n)，向上调整距离的复杂度也是 O(log n)，故删除操作的最坏情况的时间复杂度是 O(log n)。如果左偏树非常倾斜，实际应用情况下要比这个快得多。

3.6 小结

本章介绍了左偏树的各种操作，我们可以看到，左偏树作为可并堆的实现，它的各种操作性能都十分优秀，且编程复杂度比较低，可以说是一个“性价比”十分高的数据结构。左偏树之所以是很好的可并堆实现，是因为它能够捕捉到具有堆性质的二叉树里面的一些其它有用信息，没有将这些信息浪费掉。根据堆性质，我们知道，从根节点向下到任何一个外节点的路径都是有序的。存在越长的路径，说明树的整体有序性越强，与平衡树不同（平衡树根本不允许有很长的路径），左偏树尽大约一半的可能保留了这个长度，并将它甩向左侧，利用它来缩短节点的距离以提高性能。这里我们不进行严格的讨论，左偏树作为一个例子大致告诉我们：放弃已有的信息意味着算法性能上的牺牲。下面是最好的左偏树：有序表（插入操作是按逆序发生的，自然的有序性被保留了）和最坏的左偏树：平衡树（插入操作是按正序发生的，自然的有序性完全被放弃了）。



有序表　　平衡树

四、左偏树的应用

4.1 例——数字序列（Baltic 2004）

[问题描述][1]

给定一个整数序列 $a_1, a_2, \ldots, a_n$，求一个不下降序列 $b_1 \le b_2 \le \ldots \le b_n$，使得数列$\{a_i\}$和$\{b_i\}$的各项之差的绝对值之和 $|a_1 - b_1| + |a_2 - b_2| + \ldots + |a_n - b_n|$ 最小。

[数据规模] $1 \le n \le 10^6, 0 \le a_i \le 2*10^9$

[初步分析]

我们先来看看两个最特殊的情况：

1．a[1]≤a[2]≤…≤a[n]，在这种情况下，显然最优解为 b[i]=a[i]；

2．a[1]≥a[2]≥…≥a[n]，这时，最优解为 b[i]=x，其中 x 是数列 a 的中位数[2]。

于是我们可以初步建立起这样一个思路：

把 1…n 划分成 m 个区间：[q[1],q[2]-1]，[q[2],q[3]-1]，…… ，[q[m],q[m+1]-1][3]。每个区间对应一个解，b[q[i]] = b[q[i]+1] = … = b[q[i+1]-1] = w[i]，其中 w[i] 为 a[q[i]], a[q[i]+1], ... , a[q[i+1]-1] 的中位数。

显然，在上面第一种情况下 m=n，q[i]=i；在第二种情况下 m=1，q[1]=1。

这样的想法究竟对不对呢？应该怎样实现？

若某序列前半部分 a[1], a[2], … , a[n] 的最优解为 $(u,u,\ldots,u)$，后半部分 a[n+1], a[n+2], ... , a[m] 的最优解为 $(v,v,\ldots,v)$，那么整个序列的最优解是什么呢？若 u≤v，显然整个序列的最优解为 $(u,u,\ldots,u,v,v,\ldots,v)$。否则，设整个序列的最优解为 $(b[1],b[2],\ldots,b[m])$，则显然 b[n]≤u（否则我们把前半部分的解

[1] 题目来源：Baltic OI 2004 Day 1, Sequence 本文对原题略微做了改动
[2] 为了方便讨论和程序实现，本文中提到的中位数，都是指数列中第⌊ n/2⌋ 大的数
[3] 这里我们认为 q[m+1] = n+1

(b[1],b[2],…,b[n]) 改为 (u,u,…,u)，由题设知整个序列的解不会变坏），同理 b[n+1]≥v。接下来，我们将看到下面这个事实：

对于任意一个序列 a[1],a[2],…,a[n]，如果最优解为 (u,u,…,u)，那么在满足 u≤u′≤b[1] 或 b[n]≤u′≤u 的情况下，(b[1],b[2],…,b[n]) 不会比 (u′,u′,…,u′) 更优。

我们用归纳法证明 u≤u′≤b[1] 的情况，b[n]≤u′≤u 的情况可以类似证明。

当 n=1 时，u=a[1]，命题显然成立。

当 n>1 时，假设对于任意长度小于 n 的序列命题都成立，现在证明对于长度为 n 的序列命题也成立。首先把 (b[1], b[2], … b[n]) 改为 (b[1], b[1], … b[1])，这一改动将不会导致解变坏，因为如果解变坏了，由归纳假设可知 a[2],a[3],…,a[n]的中位数 w>u，这样的话，最优解就应该为(u,u,…,u,w,w,…,w)，矛盾。然后我们再把(b[1],b[1],…,b[1])改为 (u′,u′,…,u′)，由于|a[1]-x|+|a[2]-x|+…+|a[n]-x| 的几何意义为数轴上点 x 到点 a[1], a[2], … a[n] 的距离之和，且 u≤u′≤b[1]，显然点 u′ 到各点的距离之和不会比点 b[1] 到各点的距离之和大，也就是说，(b[1],b[1],…,b[n]) 不会比 (v,v,…,v) 更优。（证毕）

再回到之前的论述，由于 b[n]≤u，作为上述事实的结论，我们可以得知，将(b[1],b[2],…,b[n]) 改为(b[n],b[n],…,b[n])，再将(b[n+1],b[n+2],…,b[m]) 改为(b[n+1],b[n+1],…,b[n+1])，并不会使解变坏。也就是说，整个序列的最优解为(b[n],b[n],…,b[n],b[n+1],b[n+1],…,b[n+1])。再考虑一下该解的几何意义，设整个序列的中位数为 w，则显然令 b[n]=b[n+1]=w 将得到整个序列的最优解，即最优解为 (w,w,…,w)。

分析到这里，我们一开始的想法已经有了理论依据，算法也不难构思了。

[算法描述]

延续我们一开始的思路，假设我们已经找到前 k 个数 a[1], a[2], … , a[k] (k<n) 的最优解，得到 m 个区间组成的队列，对应的解为 (w[1],w[2], …,w[m])，现在要加入 a[k+1]，并求出前 k+1 个数的最优解。首先我们把 a[k+1] 作为一个新区间直接加入队尾，令 w[m+1]=a[k+1]，然后不断检查队尾两个区间的解 w[m] 和 w[m+1]，如果 w[m]>w[m+1]，我们需要将最后两个区间合并，并找出新区间的最优解（也就是序列 a 中，下标在这个新区间内的各项的中位数）。重复这个合并过程，直至 w[1]≤w[2]≤…≤w[m] 时结束，然后继续处理下一个数。

这个算法的正确性前面已经论证过了，现在我们需要考虑一下数据结构的选取。算法中涉及到以下两种操作：合并两个有序集以及查询某个有序集内的中位数。能较高效地支持这两种操作的数据结构有不少，一个比较明显的例子是二叉检索树(BST)，它的询问操作复杂度是 O(log n)，但合并操作不甚理想，采用启发式合并，总时间复杂度为 $O(n\log^2 n)$。

有没有更好的选择呢？通过进一步分析，我们发现，只有当某一区间内的中位数比后一区间内的中位数大时，合并操作才会发生，也就是说，任一区间与后面的区间合并后，该区间内的中位数不会变大。于是我们可以用最大堆来维护每个区间内的中位数，当堆中的元素大于该区间内元素的一半时，删除堆顶元素，这样堆中的元素始终为区间内较小的一半元素，堆顶元素即为该区间内的中位数。考虑到我们必须高效地完成合并操作，左偏树是一个理想的选择[1]。左偏树的询问操作时间复杂度为 O(1)，删除和合并操作时间复杂度都是 O(log n)，而询问操作和合并操作少于 n 次，删除操作不超过 n/2 次（因为删除操作只会在合并两个元素个数为奇数的堆时发生），因此用左偏树实现，可以把算法的时间复杂度降为 O(n log n)。

[小结]

这道题的解题过程对我们颇有启示。在应用左偏树解题时，我们往往会觉得题目无从下手，甚至与左偏树毫无关系，但只要我们对题目深入分析，加以适

[1] 前面介绍的左偏树是最小堆，但在本题中，显然只需把左偏树的性质稍做修改，就可以实现最大堆了

当的转化，问题终究会迎刃而解。这需要我们具有敏捷的思维以及良好的题感。

用左偏树解本题，相比较于前面 BST 的解法，时间复杂度和编程复杂度更低，这使我们不得不感叹于左偏树的神奇威力。这不是说左偏树就一定是最好的解法，就本题来说，解法有很多种，光是可并堆的解法，就可以用多种数据结构来实现，但左偏树相对于它们，还是有一定的优势的，这将在下一章详细讨论。

五、左偏树与各种可并堆的比较

我们知道，左偏树是一种可并堆的实现。但是为什么我们要用左偏树实现可并堆呢？左偏树相对于其他可并堆有什么优点？本章将就这个问题展开讨论，介绍各种可并堆的特点，并对它们做出比较。

5.1 左偏树的变种——斜堆

这里我们要介绍左偏树的一个变种——斜堆(Skew Heap)。斜堆是一棵堆有序的二叉树，但是它不满足左偏性质，或者说，斜堆根本就没有“距离”这个概念——它不需要记录任何一个节点的距离。从结构上来说，所有的左偏树都是斜堆，但反之不然。

类似于左偏树，斜堆的各种操作也是在合并操作的基础上完成的，因此这里只介绍斜堆的合并操作，其他操作读者都可以仿照左偏树完成。

斜堆合并操作的递归合并过程和左偏树完全一样。假设我们要合并 A 和 B 两个斜堆，且 A 的根节点比 B 的根节点小，我们只需要把 A 的根节点作为合并后新堆的根节点，并将 A 的右子树与 B 合并。由于合并都是沿着最右路径进行的，经过合并之后，新堆的最右路径长度必然增加，这会影响下一次合并的效率。为了解决这一问题，左偏树在进行合并的同时，检查最右路径节点的距离，并通过交换左右子树，使整棵树的最右路径长度非常小。然而斜堆不记录节点的距离，那么应该怎样维护最右路径呢？我们采取的办法是，从下往上，沿着合并的路径，在每个节点处都交换左右子树。

下面是斜堆合并操作的代码：

```
Function Merge(A,B)
    If A = NULL Then return B
    If B = NULL Then return A
    If key(B) < key(A) Then swap(A,B)
    right(A) ← Merge(right(A), B)
    swap(left(A), right(A))
    return A
End Function
```

斜堆的这种维护方法也是行之有效的。通过不断交换左右子树，斜堆把最右

路径甩向左边了。可以证明，斜堆合并操作的平摊时间复杂度为 O(log n)。这里略去详细的复杂度分析，感兴趣的读者可以自行参考相关的资料。

前面说过，斜堆的其他各种操作都和左偏树类似，因此斜堆各项操作的平摊时间复杂度都与左偏树相同。在空间上，由于斜堆不用记录节点的距离，因此它比左偏树的空间需求小一点。至于编程复杂度，两者都十分的低，不过斜堆的代码还是要比左偏树稍微简洁一些。至于斜堆的不足，大概可以算是它单次合并操作的时间复杂度可能退化为 O(n)，不过通常这并不影响算法的总时间复杂度。

总的来说，斜堆作为左偏树的变种，与左偏树并无优劣之分。在斜堆能派上用场的地方，左偏树同样能出色地实现算法。斜堆与左偏树之间的关系，可以类比于伸展树与 AVL 树之间的关系，只不过前两者的编程复杂度，并没有多大的差别。

5.2 左偏树与二叉堆的比较

二叉堆大概是我们最常用的一种优先队列了，它具有简洁，高效的特点，应用十分广泛。二叉堆和左偏树相比，除了合并操作外的各种操作，时间复杂度都一样，但在实际测试中，二叉堆往往比左偏树快一些。在空间上，由于二叉堆是完全二叉树，用数组表示法，可以剩下左右子树指针的空间，在这点上左偏树又吃了一亏。

介绍了二叉堆的优点，下面的这两个缺点也是不容忽视的：

1. 在进行插入、删除等操作后，二叉堆中元素的物理位置会发生改变。
2. 二叉堆的合并操作太慢，时间复杂度高达 O(n)。

当元素本身所占空间比较大时，频繁地移动元素物理位置将会影响时间效率。而有些时候，我们需要随时掌握某些元素的物理位置，以便对它进行访问或修改（比如实现删除任意已知节点操作），这时，二叉堆的第一个缺点将给我们造成一定的麻烦，我们不得不通过增加元素指针等手段来解决这个问题。而二叉堆合并效率的低下则是由其本身性质所造成的，采用启发式合并可以作为大多数情况下的一个优化，但效果仍不能令人满意。

至于二叉堆空间上的优势，也不是绝对的。数组表示法并不是在任何场合都适用的，比如当我们需要动态维护多个优先队列的时候，二叉堆不得不使用传统的指针表示法，这时二叉堆和左偏树在空间上就相差无几了。

二叉堆的上述特点，决定了它不适合作为可并堆的实现。客观的说，需要实现可并堆的题目在竞赛中并不多见，不过当我们遇到这类题目或者某些特殊的情况时，左偏树将会是比二叉堆更好的选择。

5.3 左偏树与其他可并堆的比较

前面已经提到过，可并堆可以用多种数据结构实现，左偏树并非唯一的选择。二项堆，Fibonacci 堆都是时间效率十分优秀的可并堆。

二项堆是由若干棵深度不同的二项树组成的森林。深度为 k 的二项树 B_k 由两棵二项树 B_{k-1} 连接根节点而成，有 2^k 个节点，并且满足堆性质。二项树组成二项堆的形式可以看作总节点数 n 的二进制表示形式，两个二项堆的合并可以看作二进制数的加法，这点很有启发意义。与左偏树类似，二项堆的各种操作也是在合并操作的基础上完成的。二项堆的结构和性质决定了它可以很高效地完成合并操作，与左偏树相比，虽然复杂度相同，但一般实际情况下二项堆比较快。但二项树实现取最小节点操作需要检查所有二项树的根节点，因此这项操作时间复杂度比左偏树高[1]。

Fibonacci 堆是一个很复杂的数据结构，与二项堆一样，它也是由一组堆有序的树构成的。所不同的是，Fibonacci 堆的树不一定是二项树，而且这些树是无序的，且树中兄弟节点的联系用双向循环链表来表示。Fibonacci 堆实现插入操作和合并操作只是简单地将两个 Fibonacci 堆的根表连在一起，因此这两个操作比其他可并堆都快。但在删除操作时，我们需要对 Fibonacci 堆进行维护，合并所有度数相同的树，这一步异常复杂，并且是最慢的。

有关二项堆和 Fibonacci 堆的详细介绍，有兴趣的读者可以参考相关资料，本文不再赘述。下表列出了各种可并堆的各项操作的时间复杂度[2]，已及它们的空间需求和编程复杂度。

项目	二叉堆	左偏树	二项堆	Fibonacci 堆
构建	O(n)	O(n)	O(n)	O(n)
插入	O(log n)	O(log n)	O(log n)	O(1)
取最小节点	O(1)	O(1)	O(log n)	O(1)

[1] 这里其实有一个办法可以解决这个问题，就是随时记录最小节点，并只在插入、删除以及合并等操作进行的时候更新该记录，这样二项堆的取最小节点操作时间复杂度可以降为 O(1)

[2] 表中 Fibonacci 堆的“删除最小节点”和“删除任意节点”两个操作的时间复杂度均为平摊时间复杂度，而二项堆“插入”操作的平均时间复杂度为 O(1)，表中给出的是最坏时间复杂度

删除最小节点	O(logn)	O(logn)	O(logn)	O(logn)
删除任意节点	O(logn)	O(logn)	O(logn)	O(logn)
合并	O(n)	O(logn)	O(logn)	O(1)
空间需求	最小	较小	一般	较大
编程复杂度	最低	较低	较高	很高

从表中我们可以看出，Fibonacci 堆的时间复杂度非常低，如果我们不需要进行频繁的删除操作，用 Fibonacci 堆实现可并堆将会降低算法的时间复杂度。但实际上，删除操作往往是很重要的操作，而 Fibonacci 堆的删除操作比表中其它可并堆都慢，至于它的空间需求，也大得使人望而却步。更槽糕的是 Fibonacci 堆的编程复杂度太高了，在竞赛中使用实在是不理智的行为。

至于二项堆，虽然各项操作的时间效率都十分优秀，但空间需求和编程复杂度仍然比不上左偏树。和左偏树相比，二项堆在时间效率上的优势微乎其微，用左偏树代替二项堆，的确会牺牲一些算法性能，但换来的却是简洁的代码，便于实现和调试的程序。在时间有限的竞赛中，左偏树无疑是更好的选择。

最后，我们应该认识到，虽然二项堆和 Fibonacci 堆某些操作的时间复杂度比左偏树低，但是在实际应用中，那些时间复杂度较高的操作往往会成为算法的瓶颈。比如前面的例题《数字序列》，改用二项堆或 Fibonacci 堆实现，总时间复杂度仍为 O(nlogn)，并不能起到降低算法时间复杂度的作用，实现难度反而增加了不少。

六、总结

至此，我们已经对左偏树有了深刻的认识。左偏树在时间效率上不如二项堆和 Fibonacci 堆，在空间效率上不如二叉堆，这样看来左偏树没有任何独树一帜的地方，似乎是个相当平庸的数据结构，但其实这正是左偏树的优势所在。正所谓“鱼与熊掌不可兼得”，时间复杂度、空间复杂度和编程复杂度，这三者之间很多时候是矛盾的。Fibonacci 堆时间复杂度最低，但编程复杂度让人无法接受；二叉堆的空间复杂度和编程复杂度都很低，但时间复杂度却是它的致命弱点。左偏树很好地协调了三者之间的矛盾，并且在存储性质上，没有二叉堆那样的缺陷，因此左偏树的适用范围十分广。左偏树不但可以高效方便地实现可并堆，更可以作为二叉堆的替代品，应用于各种优先队列，很多时候甚至比二叉堆更方便。

【感谢】

感谢余江伟同学及阮志远老师的热心帮助和修改意见。

【参考文献】

[1] **傅清祥，王晓东　算法与数据结构（第二版）　电子工业出版社**

[2] **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein Introduction to Algorithms (Second Edition)　The MIT Press**

[3] **Mark Allen Weiss　Data Structures and Algorithm Analysis in C　(Second Edition)　Pearson Education**

【附录】

附录 I：例题《数字序列》的原题

SEQUENCE

Short formulation. The number sequence is given. Your task is to construct the increasing sequence that approximates the given one in the best way. The best approximating sequence is the sequence with the least total deviation from the given sequence.

More precisely. Let $t_1, t_2, \ldots, t_N$ is the given number sequence. Your task is to construct the increasing number sequence $z_1 < z_2 < \ldots < z_N$.

The sum $|t_1 - z_1| + |t_2 - z_2| + \ldots + |t_N - z_N|$ should be a minimal feasible.

Input

There is the integer N ($1 \le N \le 1000000$) in the first line of input file `seq.in`. Each of the next N lines contains single integer – the given sequence element. There is t_K in the (K+1)-th line. Any element is satisfying to relation $0 \le t_K \le 2000000000$.

Output

The first line of output file `seq.out` must contain the single integer – the minimal possible total deviation. Each of the next N lines must contain single integer – the recurrent element of the best approximating sequence.

If there are several solutions, your program must output any one sequence with a least total deviation.

Example

seq.in	seq.out
7 9 4 8 20 14 15 18	13 6 7 8 13 14 15 18

附录 II：例题《数字序列》左偏树解法的程序

```
PROGRAM Seq_LeftistTree;
TyPe
  node=
  record
    dist:byte;
    key,left,right:longint;
  end;
VaR
  nd:array[0..1000000]of node;
  stk,q:array[0..1000000]of longint;
  n,cl,i:longint;
Function merge(a,b:longint):longint;
  var
    c:longint;
  begin
    if (a=0)or(b<>0)and(nd[a].key<nd[b].key) then
    begin
      c:=a;
      a:=b;
      b:=c;
    end;
    merge:=a;
    if b=0 then exit;
    nd[a].right:=merge(nd[a].right,b);
    if nd[nd[a].left].dist<nd[nd[a].right].dist then
    begin
      c:=nd[a].left;
      nd[a].left:=nd[a].right;
      nd[a].right:=c;
    end;
    nd[a].dist:=nd[nd[a].right].dist+1;
  end;
Procedure print;
  var
    tot,i,j:longint;
  begin
    tot:=0;
    for i:=1 to cl do
      for j:=q[i-1]+1 to q[i] do
        inc(tot,abs(nd[j].key-nd[stk[i]].key));
    writeln(tot);
```

```
  end;
BeGiN
  assign(input,'seq.in');
  reset(input);
  assign(output,'seq.out');
  rewrite(output);
  readln(n);
  fillchar(nd,sizeof(nd),0);
  cl:=0;
  q[0]:=0;
  for i:=1 to n do
  begin
    readln(nd[i].key);
    dec(nd[i].key,i);
    inc(cl);
    stk[cl]:=i;
    q[cl]:=i;
    while (cl>1)and(nd[stk[cl]].key<nd[stk[cl-1]].key) do
    begin
      dec(cl);
      stk[cl]:=merge(stk[cl],stk[cl+1]);
      if odd(q[cl+1]-q[cl]) and odd(q[cl]-q[cl-1]) then
        stk[cl]:=merge(nd[stk[cl]].left,nd[stk[cl]].right);
      q[cl]:=q[cl+1];
    end;
  end;
  print;
  close(output);
EnD.
```