随机平衡二叉查找树 Treap 的分析与应用

郭家宝 2010011267

清华大学计算机科学与技术系 02 班 北京 100084

http://www.byvoid.com

摘要

Treap 是一种实现简洁，时空高效的平衡二叉查找树。本文分析了 Treap 的平衡性，详述了 Treap 的构造方法，与其他各种平衡树进行了比较，并介绍了 Treap 在算法设计中的应用。

关键词

Treap 二叉查找树 平衡树 数据结构 动态统计

引言

本文写作的目的

Treap 作为被广泛应用的一种平衡树数据结构，一直以来许多人都在研究或想要研究。但笔者经历多番查找资料，很难找到一个关于 Treap 的系统性、总结性而又简明易懂的论文。本文想通过对 Treap 的介绍，起到抛砖引玉的作用。欢迎大家指正和批评。

阅读本文所需掌握的预备知识

- ✓ 基础的数学知识
- ✓ 计算机程序设计
- ✓ 基本算法
- ✓ 基本数据结构

目录

一、序言

1. 我们为什么要排序

Treap 是用来**排序**(Sort)的一种**数据结构**(Data Structure)。在讨论 Treap 之前，让我们先讨论一下，我们为什么要排序？当我们看到世间万物的时候，是否想探究其内在的规律，是否想了解自然的顺序？也许你认为，排序当然是必要的，我给出我认为需要排序的三点理由：

1. **有序的事物符合人类大脑结构**
   你可以轻易地一眼看清少于 6 个物体，更多的话就需要数了。在有序的事物中，你可以很快得找到需要的信息，因为**有序的事物符合人类大脑的认识规律**，内在的排序不是人类的本能。

2. **有序的事物符合数学规律**
   有序的元素之间有着递增或递减的单调性。对于探究元素的内在联系，排序是极其重要的。

3. **打乱顺序是容易的，建立顺序是困难的**
   如热力学第二定律指出，对不可逆过程，系统的熵总是增加的。也就是元素就是自发的走向无序的状态，建立顺序是需要代价的，不能自发的进行，所以需要我们主动来排序。

2. 基于比较的排序

在计算机科学中，排序是一门基础的算法技术，许多算法都要以此作为基础，不同的排序算法有着不同的时间开销和空间开销。从 1959 年 Donald Shell 发明了冲破 $O(N^2)$时间屏障的希尔排序，到 1962 年 C. A. R. Hoare 发明了时间复杂度为 O(NlogN)快速排序，排序已经被认为是一个已解决的问题。然而至今新的排序算法依然在不断产生，如 2005 年被发明的图书馆排序。排序算法已经有非常多种了，最直观的，序列中数据之间需要进行比较的排序，被称为**基于比较的排序**。

(1) 基于比较的排序的三种手段

在基于比较的排序算法中，存在着两种基本的思想，即对于混乱的数据，**调整**或者**重建**。于是有三种基本手段：**交换**、**选择**、**插入**。

交换是实现调整的最直接手段，基本方法是把逆序的数据进行交换，以实现顺序排列。以重建实现排序，考虑是每次从原序列中挑出应该取的元素，还是从原序列中顺序地取出元素，

考虑插入到新序列中哪个位置。前者的手段为**选择**，后者的手段是**插入**。

在基于交换的排序算法中，人们最早发明了 $O(N^2)$的**冒泡排序**(Bubble Sort)。这种算法十分简单，而且只需要常数的额外的空间，所有一直以来是初学者一定要了解的排序算法。后来有人发明了基于交换的**快速排序**(Quick Sort)，它的平均时间界为 O(NlogN)。迄今为止，快速排序是一般实际应用中效果最好的算法。

在基于选择的排序算法中，人们最早发明了 $O(N^2)$的**选择排序**(Selection Sort)。选择排序的算法更为容易理解和实现，就是每次都从原序列中顺序查找出最小的元素，放入新的序列的下一个位置。在这种思想的引导下**堆排序**(Heap Sort) 被发明了。与选择排序一样，对排序仍然是每次从原序列中取出最小的元素，放在新序列的下一个位置。但是不同在于使用了堆这一数据结构，取出最小元素仅需要 O(LogN)的时间，于是它的时间复杂度为 O(NlogN)。

我们玩扑克牌的时候，习惯于每起得一张牌后，直接将它是插入到合适的位置，以实现顺序的排列。**插入排序**(Insertion Sort)正是运用了这种思想，方法是从原序列中取出下一个元素，然后再新序列中找到合适的位置并插入。使用线性表存储时，由于新序列中已有的元素是已经有序的，确定插入的位置时，可以使用**二分查找**(Binary Search)，使时间降为 O(LogN)。但是不同于扑克牌，在线性表中，我们只能把这个位置后面的元素一个个向后移动，才能插入这个元素，最坏情况下可能会移动所有的元素，时间为 O(N)。为了加快插入，我们可以使用链表，这样插入的时间复杂度无论如何就变成了 O(1)了，但是确定插入的合适的位置却变成了 O(N)。总而言之无论是基于线性表还是基于链表的简单插入排序，时间复杂度都是 $O(N^2)$。为了解决这种矛盾，人们发明了**跳跃表**(Skip List)和**二叉查找树**(Binary Search Tree)，基于这两种数据结构的排序的期望时间复杂度都是 O(NlogN)。

3. 二叉查找树

二叉查找树(Binary Search Tree)是基于插入思想的一种**在线的**排序数据结构。它又叫二叉搜索树(Binary Search Tree)、二叉排序树(Binary Sort Tree)，简称 BST。这种数据结构的基本思想是在二叉树的基础上，规定一定的顺序，使数据可以有序地存储。二叉查找树运用了像二分查找一样的查找方式，并且基于链式结构存储，从而实现了高效的查找效率和完美的插入时间。

二、二叉查找树

1. 二叉查找树的定义、遍历与查找

(1) 定义

二叉查找树(Binary Search Tree)或者是一棵空树，或者是具有下列性质的二叉树：

1. 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
2. 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
3. 它的左、右子树也分别为二叉查找树。

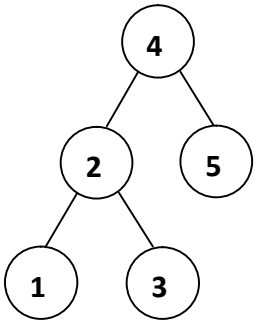

图 1

上述性质被称为 **BST 性质**。可以看出，二叉查找树是递归定义的，如图 1 是一个二叉查找树。

代码 1 给出了一个 BST 节点的定义。

```
struct BST_Node
{
    BST_Node *left,*right;//节点的左右子树的指针
    int value; //节点的值
};
```

代码 1

(2) 遍历

对于一个已知的二叉查找树，从小到大输出其节点的值，只需对其进行二叉树的**中序遍历**，即递归地先输出其左子树，再输出其本身，然后输出其右子树。遍历的时间复杂度为 O(N)。

代码 2 为把一个以 root 为根的 BST 所有节点的值从小到大输出到标准输出流。

```
BST_Node *root;

void BST_Print(BST_Node *P)
{
    if (P)
    {
        BST_Print(P->left);
        printf("%d\n",P->value);
        BST_Print(P->right);
    }
}

int main()
{
    BST_Print(root);
    return 0;
}
```

代码 2

(3) 查找

对于一个已知的二叉查找树，在其中查找特定的值，方法如下。

1. 从根节点开始查找；
2. 如果当前节点的值就是要查找的值，查找成功；
3. 如果要查找的值小于当前节点的值，在当前节点的左子树中查找该值；
4. 如果要查找的值大于当前节点的值，在当前节点的右子树中查找该值；
5. 如果当前节点为空节点，查找失败，二叉查找树中没有要查找的值。

查找的期望时间复杂度为 O(logN)。

代码 3 为在一个已知的二叉查找树中查找值 5。

```
BST_Node *root;

BST_Node *BST_Find(BST_Node *P,int value)
{
    if (!P)
        return 0; //查找失败
    if (P->value==value)
        return P; //查找成功
    else if (value < P->value)
        return BST_Find(P->left,value);   //在左子树中查找
    else
        return BST_Find(P->right,value); //在右子树中查找
}

int main()
{
    BST_Node *result;
    result=BST_Find(root,5);//在 BST 中查找值为 5 的节点
    if (result)
        printf("Found!");
    else
        printf("Not Found!");
    return 0;
}
```

代码 3

2. 二叉查找树的插入与删除

(1) 插入

在二叉查找树中插入元素，要建立在查找的基础上。基本方法是类似于线性表中的二分查找，不断地在树中缩小范围定位，最终找到一个合适的位置插入。具体方法如下所述。

1. 从根节点开始插入；
2. 如果要插入的值小于等于当前节点的值，在当前节点的左子树中插入；
3. 如果要插入的值大于当前节点的值，在当前节点的右子树中插入；
4. 如果当前节点为空节点，在此建立新的节点，该节点的值为要插入的值，左右子树为空，插入成功。

对于相同的元素，一种方法我们规定把它插入左边，另一种方法是我们在节点上再加一个域，记录重复节点的个数。上述方法为前者。插入的期望时间复杂度为 O(logN)。

代码 4 为在一个二叉查找树中插入值 5。

```
BST_Node *root;

BST_Node *BST_Insert(BST_Node *&P,int value) //节点指针要传递引用
{
    if (!P)
    {
        P=new BST_Node; //插入成功
        P->value=value;
    }
    else if (value <= value)
        return BST_Insert(P->left,value);   //在左子树中插入
    else
        return BST_Insert(P->right,value); //在右子树中插入
}

int main()
{
    BST_Insert(root,5);//在 BST 中插入值 5
    return 0;
}
```

代码 4

(2) 删除

二叉查找树的删除稍有些复杂，要分三种情况分别讨论。基本方法是要先在二叉查找树中找到要删除的节点的位置，然后根据节点分以下情况：

情况一，该节点是**叶节点**(没有非空子节点的节点)，直接把节点删除即可。

情况二，该节点是**链节点**(只有一个非空子节点的节点)，为了删除这个节点而不影响它的子树，需要把它的子节点代替它的位置，然后把它删除。如图 2 所示，删除节点 2 时，需要把它的左子节点代替它的位置。

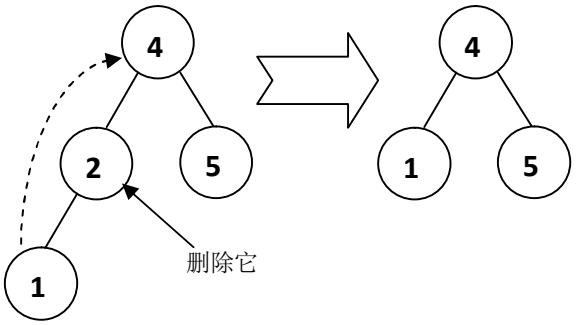

图 2

情况三，该节点有两个非空子节点。由于情况比较复杂，一般的策略是用它右子树的最小值来代替它，然后把它删除。如图 3 所示，删除节点 2 时，在它的右子树中找到最小的节点 3，该节点一定为待删除节点的**后继**。删除节点 3(它可能是叶节点或者链节点)，然后把节点 2 的值改为 3。

实际上在实现的时候，情况一和情况二是可以一样对待的，因为用叶节点的子节点代替它本身，就是用空节点代替了它，等同于删除。对待情况三除了可以用后继代替本身以外，我们还可以使用它的前驱(左子树的最大值)代替它本身。

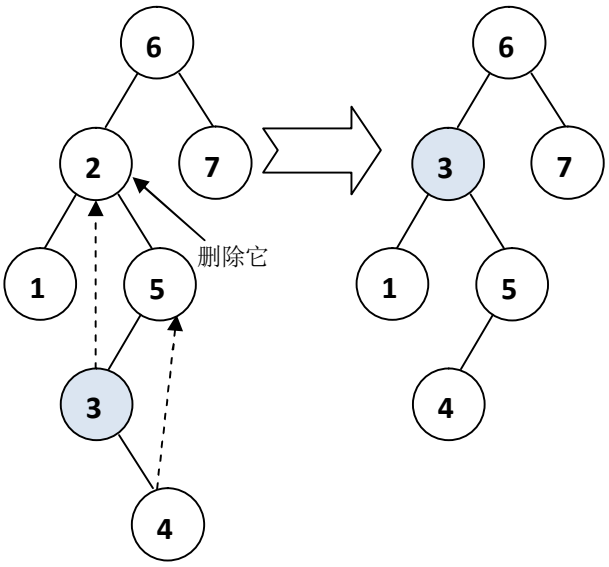

图 3

牵涉到的查找最小值的问题，章节三.4.(4)有详细讨论。基本方法就是从子树的根节点开始，如果左子节点不为空，那么就访问左子节点，直到左子节点为空，当前节点就是该子树的最小值节点。删除它只需用它的右子节点代替它本身。

代码 5 给出了在一个给定的二叉查找树中删除值 2 的完整代码。

```
BST_Node *root;

int Find_DeleteMin(BST_Node *&P)//返回子树的最小值并删除 节点指针要传递引用
{
    if (!P->left) //如果已经没有左子节点，删除它
    {
        BST_Node *t=P;
        int r=P->value;
        P=P->right;
        delete t;
        return r;
    }
    else
        return Find_DeleteMin(P->left); //如果还有左子节点，访问左子节点
}

void BST_Delete(BST_Node *&P,int value)//节点指针要传递引用
{
    if (!P)
        return ; //查找失败，BST 中不存在要删除的节点
    if (P->value==value) //查找成功，对其删除
    {
        if (P->left && P->right)
            P->value=Find_DeleteMin(P->right);
        else
        {
            BST_Node *t=P;
            if (P->left)
                P=P->left;
            else
                P=P->right;
            delete t;
        }
    }
    else if (value < P->value)
        BST_Delete(P->left,value);   //在左子树中查找
    else
        BST_Delete(P->right,value); //在右子树中查找
}


int main()
{
```

```
    BST_Delete(root,2);//在 BST 中删除值 2
    return 0;
}
```

代码 5

3. 二叉查找树的平衡性问题讨论

二叉查找树的效果究竟如何？事实上，随机的进行 N^2(N>=1000)次插入和删除之后，二叉查找树会趋向于向左偏沉。为什么会出现这种情况，原因在于删除时，我们总是选择将待删除节点的后继代替它本身。这样就会造成总是右边的节点数目减少，以至于树向左偏沉。已经被证明，随机插入或删除 N^2 次以后，树的期望深度为 $\Theta(N^{1/2})$。我们可以在删除时随机地选择用前驱还是后继代替本身，以消除向一边偏沉的问题。这种神奇地做法消除了偏沉的不平衡，效果十分明显。

对待随机的数据二叉查找树已经做得很不错了，但是如果有像这样 6,5,4,3,2,1 有序的数据插入树中时，会有什么后果出现？如图 4。二叉查找树退化成为了一条链。这个时候，无论是查找、插入还是删除，都退化成了 O(N)的时间。

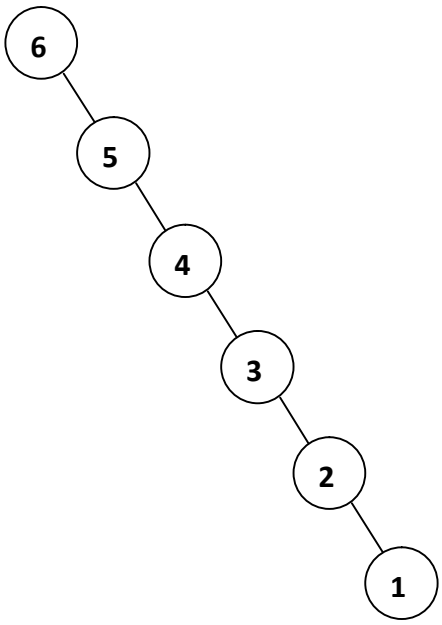

图 4

事实上，在实际的应用中，这种情况会经常出现[1]，而简单的二叉查找树却没有办法变得更快。我们需要使二叉查找树变得尽量平衡，才能保证各种操作在 O(logN)的期望时间内完成，于是各种**自动平衡二叉查找树**(Self-Balancing Binary Search Tree)因而诞生。

在常见的自动平衡二叉查找树(以下简称**平衡树**)中，有近乎完美平衡的 AVL 树(Adelson-Velskii

[1] 例如在搜索时存储某些按照一定顺序产生的状态时。

& Landis Tree)，红黑树(Red Black Tree)和 SBT(Size Balanced Tree)等，以及期望平衡的伸展树(Splay Tree)，和 Treap(Tree & Heap)等数据结构。

Treap 具有可观的平衡性，且最易于编码调试等特点，因此在信息学竞赛中被广泛地使用。

三、 Treap

1. 什么是 Treap

(1) Treap = Tree + Heap

Treap 是一种平衡树。Treap 发音为[tri:p]。这个单词的构造选取了 Tree(树)的前两个字符和 Heap(堆)的后三个字符，Treap = Tree + Heap。顾名思义，Treap 把 BST 和 Heap 结合了起来。它和 BST 一样满足许多优美的性质，而引入堆目的就是为了维护平衡。

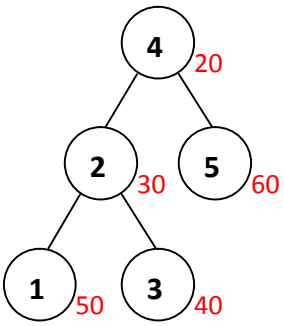

图 5

Treap 在 BST 的基础上，添加了一个**修正值**。在满足 BST 性质的基础上，Treap 节点的修正值还满足**最小堆性质**[2]。最小堆性质可以被描述为每个子树根节点都小于等于其子节点。于是，Treap 可以定义为有以下性质的二叉树：

1. 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值，而且它的根节点的修正值小于等于左子树根节点的修正值；
2. 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值，而且它的根节点的修正值小于等于右子树根节点的修正值；
3. 它的左、右子树也分别为 Treap。

图 5 为一个 Treap。

修正值是节点在插入到 Treap 中时**随机**生成的一个值，它与节点的值无关。代码 6 给出了 Treap 的一般定义。

```
struct Treap_Node
{
    Treap_Node *left,*right; //节点的左右子树的指针
    int value,fix; //节点的值和修正值
};
```

代码 6

[2] 修正值全部满足最大堆性质也是可以的，在本文的介绍中，修正值全部是满足最小堆性质的。

(2) 为什么平衡

我们发现，BST 会遇到不平衡的原因是因为有序的数据会使查找的路径退化成链，而随机的数据使 BST 退化的概率是非常小的。在 Treap 中，修正值的引入恰恰是使树的结构不仅仅取决于节点的值，还取决于修正值的值。然而修正值的值是随机生成的，出现有序的随机序列是小概率事件，所以 Treap 的结构是趋向于随机平衡的。

2. 如何构建 Treap

(1) 旋转

为了使 Treap 中的节点同时满足 BST 性质和最小堆性质，不可避免地要对其结构进行调整，调整方式被称为**旋转**。在维护 Treap 的过程中，只有两种旋转[3]，分别是左旋转(简称**左旋**)和右旋转(简称**右旋**)。

旋转是相对于子树而言的，左旋和右旋的命名体现了旋转的一条性质：

旋转的性质 1
左旋一个子树，会把它的根节点旋转到根的左子树位置，同时根节点的右子节点成为子树的根；右旋一个子树，会把它的根节点旋转到根的右子树位置，同时根节点的左子节点成为子树的根。

如图 6 所示，我们可以从图中清晰地看出，左旋后的根节点降到了左子树，右旋后根节点降到了右子树，而且仍然满足 BST 性质，于是有：

旋转的性质 2
对子树旋转后，子树仍然满足 BST 性质。

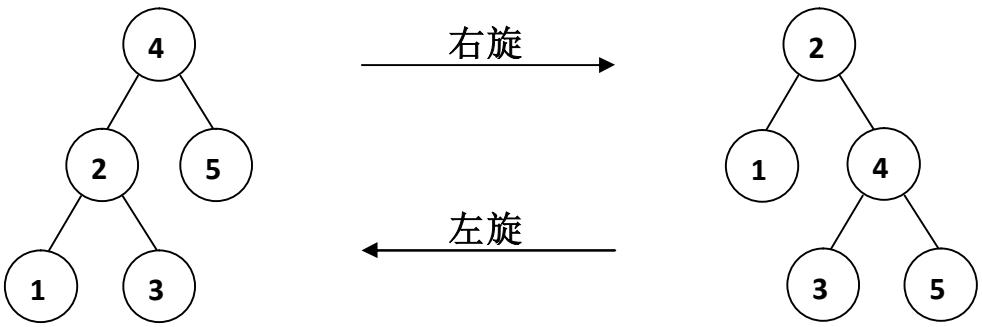

图 6

[3] 其他的平衡树如 AVL 有更多的旋转方式。

利用旋转的两条重要性质，我们可以来改变树的结构，实际上我们恰恰是通过旋转，使 Treap 节点之间满足堆序。

如图 7 所示的左边的一个 Treap，它仍然满足 BST 性质。但是由于某些原因，节点 4 和节点 2 之间不满足最小堆序，4 作为 2 的父节点，它的修正值大于左子节点的修正值。我们只有将 2 变成 4 的父节点，才能维护堆序。根据旋转的性质我们可以知道，由于 2 是 4 的左子节点，为了使 2 成为 4 的父节点，我们需要把以 4 为根的子树右旋。右旋后，2 成为了 4 的父节点，满足堆序。

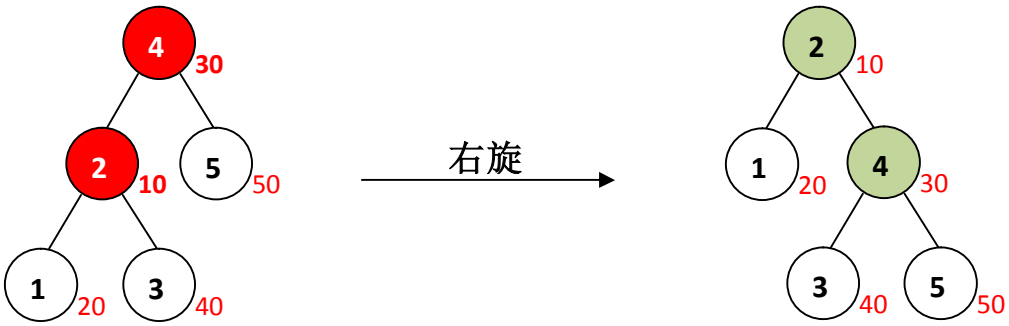

图 7

由此我们可以总结出，**旋转的意义**在于：
旋转可以使不满足堆序的两个节点通过调整位置，重新满足堆序，而不改变 BST 性质。

代码 7 给出了两种旋转的实现。

```
void Treap_Left_Rotate(Treap_Node *&a) //左旋  节点指针一定要传递引用
{
    Treap_Node *b=a->right;
    a->right=b->left;
    b->left=a;
    a=b;
}

void Treap_Right_Rotate(Treap_Node *&a) //右旋  节点指针一定要传递引用
{
    Treap_Node *b=a->left;
    a->left=b->right;
    b->right=a;
    a=b;
}
```

代码 7

(2) 遍历和查找

像大多是平衡树一样[4]，在 Treap 中查找和遍历不会改变 Treap 的结构。所以在 Treap 中查找和遍历的方法，与基本的二叉查找树完全相同。具体方法参见二.1.(2)和二.1.(3)。

(3) 插入

在 Treap 中插入元素，与在 BST 中插入方法相似。首先找到合适的插入位置，然后建立新的节点，存储元素。但是要注意建立新的节点的过程中，会随机地生成一个修正值，这个值可能会破坏堆序，因此我们要根据需要进行恰当的旋转。具体方法如下：

1. 从根节点开始插入；
2. 如果要插入的值小于等于当前节点的值，在当前节点的左子树中插入，插入后如果左子节点的修正值小于当前节点的修正值，对当前节点进行右旋；
3. 如果要插入的值大于当前节点的值，在当前节点的右子树中插入，插入后如果右子节点的修正值小于当前节点的修正值，对当前节点进行左旋；
4. 如果当前节点为空节点，在此建立新的节点，该节点的值为要插入的值，左右子树为空，插入成功。

举例说明，如图 8，在已知的 Treap 中插入值为 4 的元素。找到插入的位置后，随机生成的修正值为 15。

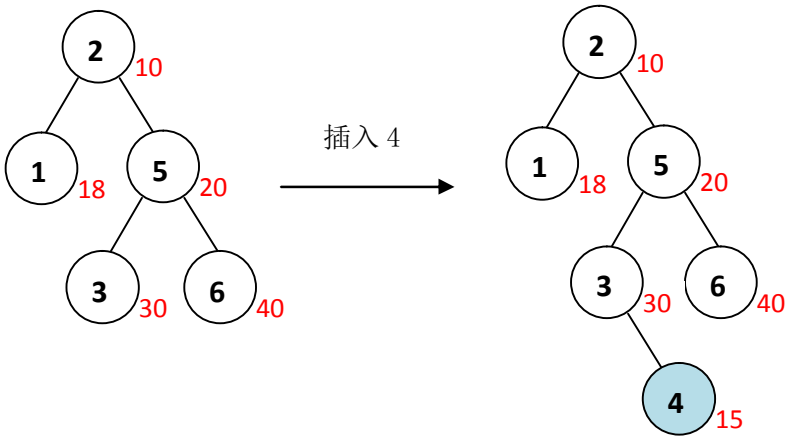

图 8

新建的节点 4 与他的父节点 3 之间不满足堆序，对以节点 3 为根的子树左旋，如图 9。

[4] 伸展树(Splay)等少数除外。

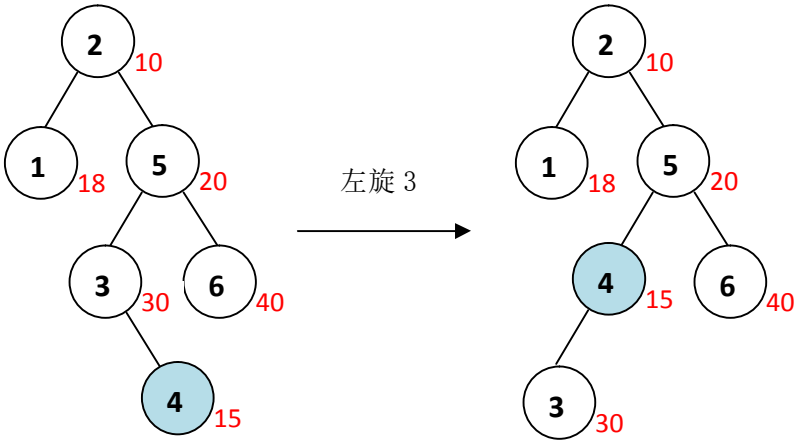

图 9

节点 4 与其父节点 5 仍不满足最小堆序，对以节点 5 为根的子树右旋，如图 10。

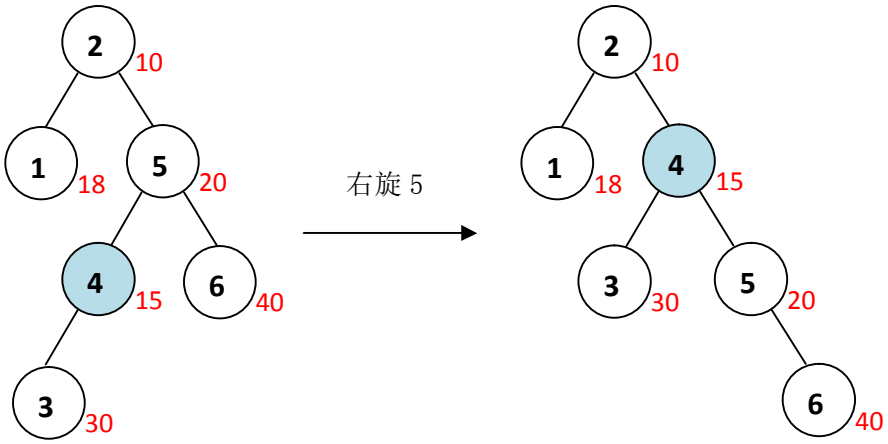

图 10

至此，节点 4 与其父亲 2 满足堆序，调整结束。

在 Treap 中插入元素的期望时间是 O(logN)。代码 8 为在 Treap 中插入一个值为 7 的元素的代码。

```
Treap_Node *root;

void Treap_Insert(Treap_Node *&P,int value) //节点指针一定要传递引用
{
    if (!P) //找到位置，建立节点
    {
        P=new Treap_Node;
        P->value=value;
        P->fix=rand();//生成随机的修正值
    }
    else if (value <= P->value)
    {
        Treap_Insert(P->left,r);
        if (P->left->fix < P->fix)
```

```
            Treap_Right_Rotate(P);//左子节点修正值小于当前节点修正值，右旋当前节点
    }
    else
    {
        Treap_Insert(P->right,r);
        if (P->right->fix < P->fix)
            Treap_Left_Rotate(P);//右子节点修正值小于当前节点修正值，左旋当前节点
    }
}

int main()
{
    Treap_Insert(root,7);//在 Treap 中插入值为 7 的元素
    return 0;
}
```

代码 8

(4) 删除

与 BST 一样，在 Treap 中删除元素要考虑多种情况。我们可以按照在 BST 中删除元素同样的方法来删除 Treap 中的元素，即用它的后继(或前驱)节点的值代替它，然后删除它的后继(或前驱)节点。为了不使 Treap 向一边偏沉，我们需要随机地选取是用后继还是前驱代替它，并保证两种选择的概率均等。

上述方法期望时间复杂度为 O(logN)，但是这种方法并没有充分利用 Treap 已有的随机性质，而是重新得随机选取代替节点。我们给出一种更为通用的删除方法，这种方法是基于旋转调整的。首先要在 Treap 树中找到待删除节点的位置，然后分情况讨论：

情况一，该节点为叶节点或链节点，则该节点是**可以直接删除的节点**。若该节点有非空子节点，用非空子节点代替该节点的，否则用空节点代替该节点，然后删除该节点。

情况二，该节点有两个非空子节点。我们的策略是通过旋转，使该节点变为可以直接删除的节点。如果该节点的左子节点的修正值小于右子节点的修正值，右旋该节点，使该节点降为右子树的根节点，然后访问右子树的根节点，继续讨论；反之，左旋该节点，使该节点降为左子树的根节点，然后访问左子树的根节点，继续讨论，知道变成可以直接删除的节点。

下面给一个删除例子：在 Treap 中删除值为 6 的元素。如图 11，首先在 Treap 中找到 6 的位置。

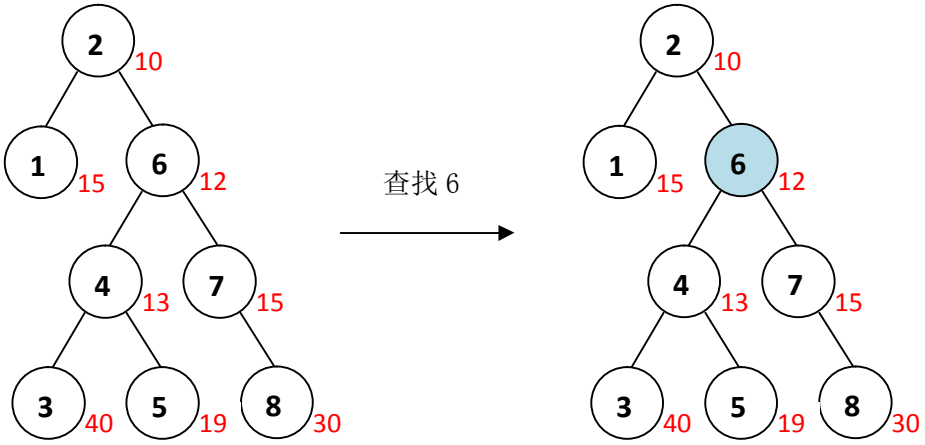

图 11

发现节点 6 有两个子节点，且左子节点的修正值小于右子节点的修正值，需要右旋节点 6，如图 12。

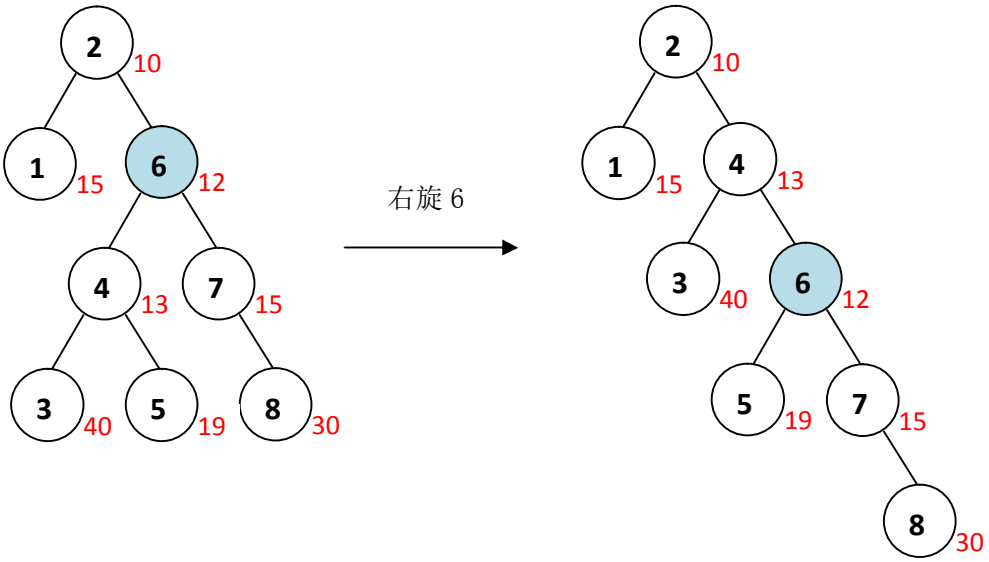

图 12

旋转后，节点 6 仍有两个节点，右子节点修正值较小，于是左旋节点 6，如图 13。

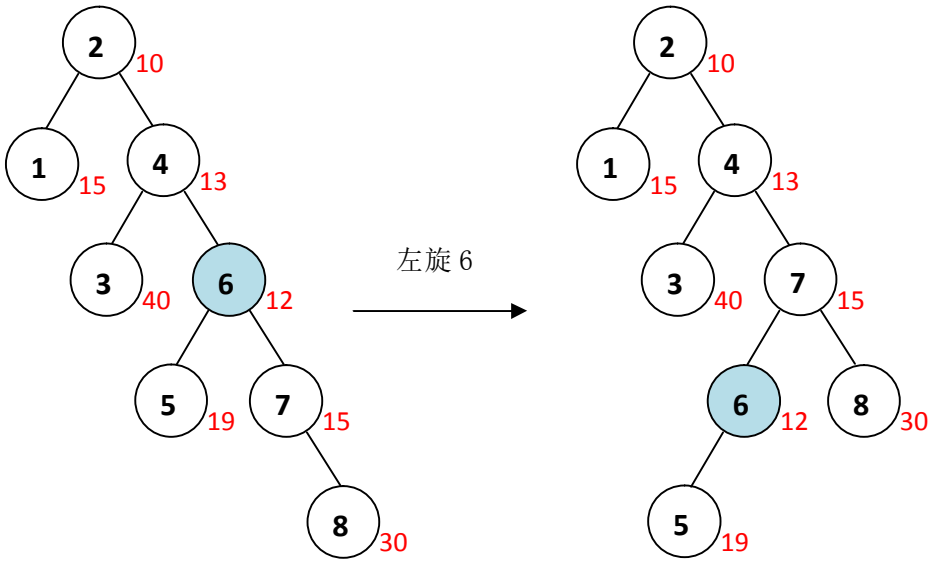

图 13

此时，节点 6 只有一个子节点，可以直接删除，用它的左子节点代替它，删除本身，如图 14。

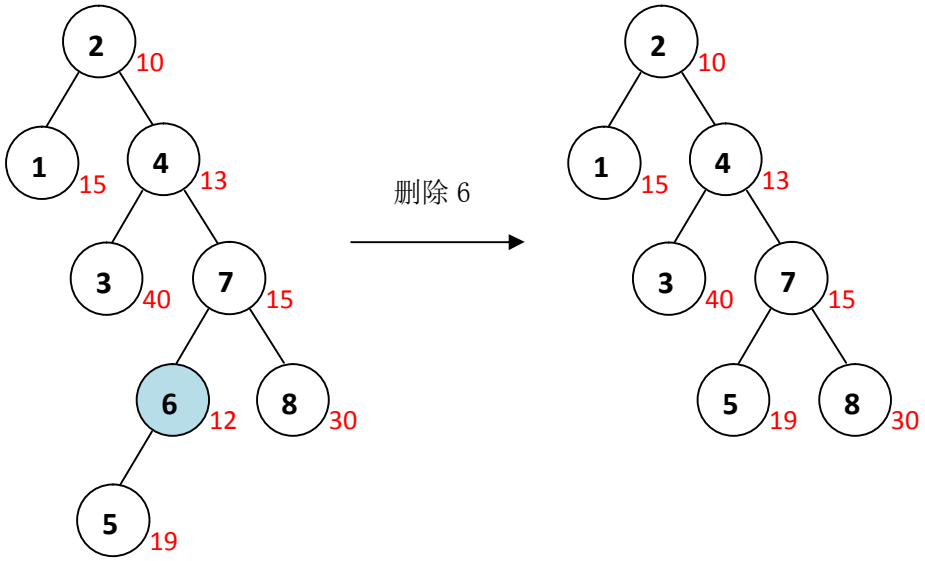

图 14

删除的复杂度稍高，但是期望时间仍为 O(logN)，但是在程序中更容易实现。代码 9 给出了后一种(即上述图例中)的删除方法，在给定的 Treap 中删除值为 6 的节点的代码。

```
BST_Node *root;

void Treap_Delete(Treap_Node *&P,int *value) //节点指针要传递引用
{
    if (value==P->value) //找到要删除的节点 对其删除
    {
        if (!P->right || !P->left) //情况一，该节点可以直接被删除
        {
```

```
            Treap_Node *t=P;
            if (!P->right)
                P=P->left;   //用左子节点代替它
            else
                P=P->right; //用右子节点代替它
            delete t; //删除该节点
        }
        else //情况二
        {
            if (P->left->fix < P->right->fix) //左子节点修正值较小，右旋
            {
                Treap_Right_Rotate(P);
                Treap_Delete(P->right,r);
            }
            else   //左子节点修正值较小，左旋
            {
                Treap_Left_Rotate(P);
                Treap_Delete(P->left,r);
            }
        }
    }
    else if (value < P->value)
        Treap_Delete(P->left,r);   //在左子树查找要删除的节点
    else
        Treap_Delete(P->right,r); //在右子树查找要删除的节点
}

int main()
{
    Treap_Delete(root,6);//在 Treap 中删除值为 6 的元素
    return 0;
}
```

代码 9

3. 为什么要用 Treap

(1) Treap 的特点

1. Treap **简明易懂**。Treap 只有两种调整方式，左旋和右旋。而且即使没有严密的数学证明和分析，Treap 的构造方法啊，平衡原理也是不难理解的。只要能够理解 BST 和堆的思想，理解 Treap 当然不在话下。
2. Treap **易于编写**。Treap 只需维护一个满足堆序的修正值，修正值一经生成无需修改。相

比较其他各种平衡树，Treap 拥有最少的调整方式，仅仅两种相互对称的旋转。所以 Treap 当之无愧是最易于编码调试的一种平衡树。

3. Treap **稳定性佳**。Treap 的平衡性虽不如 AVL，红黑树，SBT 等平衡树，但是 Treap 也不会退化[5]，可以保证期望 O(logN)的深度。Treap 的稳定性取决于随机数发生器。
4. Treap 具有**严密的数学证明**。Treap 期望 O(logN)的深度，是有严密的数学证明的。但这不是本文介绍的重点，大多略去。
5. Treap 具有**良好的实践效果**。各种实际应用中，Treap 的稳定性表现得相当出色，没有因为任何的构造出的数据而退化。于是在信息学竞赛中，不少选手习惯于使用 Treap，均取得了不俗的表现。

(2) Treap 与其他平衡树的比较

与 BST 相比[6]：

显而易见的，BST 更加容易编程实现。对于完全随机的数据，BST 会比 Treap 更快，因为 BST 没有旋转等操作。但是在实际的应用中，往往会存在大量有序的数据，这时 BST 会退化，而 Treap 仍旧能够保持随机的平衡。

与 Splay 相比：

Splay 和 BST 一样，不需要维护任何附加域，比 Treap 在空间上有节约。Splay 伸展操作中要用到的旋转相对于 Treap 要稍复杂，编程实现不如 Treap 容易。而且 Splay 在查找时也会调整结构，这使得 Splay 灵活性稍有欠缺。Splay 的查找插入删除等基本操作的时间复杂度为均摊 O(logN)而非期望。可以故意构造出使 Splay **变得很慢**的数据，这在信息学竞赛中是很不利的。Splay 找到了一个时间、空间和编程效率上的平衡点。

与 AVL，红黑树相比：

AVL，红黑树的平衡性是严格的，稳定性表现得十分出色。与 Treap 一样，它们都要维护附加的域(高度、颜色)来实现平衡。AVL 和红黑树在调整的过程中，旋转都是均摊 O(1)的，而 Treap 要 O(logN)。与 Treap 的随机修正值不同，它们维护的附加域要动态的调整，而 Treap 的随机修正值一经生成不再改变，这一点使得灵活性不如 Treap。最重要的是，AVL 和红黑树都是时间效率很高的经典算法，在许多专业的应用领域(如 STL)有着十分重要的地位。然而 AVL 和红黑树的编程实现的难度要比 Treap 大得多，正是由于过于复杂的编程，使得它们在信息学竞赛中备受冷落。

与 SBT 相比：

作为平衡树中的新秀[7]，SBT 有着能与 AVL 和红黑树相媲美的严格平衡性，而实现的难度却远小于 AVL 和红黑树。SBT 的平衡附加域是子树的大小，而非其他的“无用”的值。SBT 十分简洁高效，灵活性也很优秀。编程实现的难度要稍大于 Treap。然而 SBT 没有受到学术界重视，原因是因为它只是在 AVL 的基础上进行常数级的优化，而并没有突破性的进展[8]。

表格 1 为几种平衡树的各种特性的对比。

[5] 如果随机函数是正常的，Treap 的退化是小概率事件。
[6] 确切地说，简单的 BST 不是平衡树，放在这里仅仅是与 Treap 比较。
[7] SBT(Size Balanced Tree)是中国广东中山纪念中学的高中学生陈启峰在 2006 年发明的一种平衡树。
[8] 按照 dd_engi 的说法，其中一个原因是因为 SBT 的发明者与 AVL 相比晚生几十年。

平衡树	附加域	平衡性	运行效率	编程难度	实用性[9]	特性
Treap	修正值	较好	较快	易	好	随机平衡
BST	无	差	不稳定	易	一般	编写容易
Splay	无	-	中	中	好	灵活易变
AVL	子树高度	好	快	难	较差	经典算法
红黑树	节点颜色	好	快	难	较差	效率极佳
SBT	子树大小	好	快	中	好	短小精悍

表格 1

4. Treap 的更多操作与技巧

查找、插入、删除是平衡树最基本的三种操作，但是在实际的应用中许多其他的操作都是必要的，而且 Treap 这种强大的数据结构的功能远远不止此，下面我们要讨论的是 Treap 更多的操作，以及一些技巧。

(1) 懒惰删除

基本的删除操作，比起插入和查找要稍有复杂。有时候，我们不愿意再写一段删除的程序代码，于是采用了**懒惰删除**(lazy deletion)的方法。

懒惰删除就是在删除时，仅仅将元素找到后给元素打上“已被删除”的标记，而实际上不把它从平衡树中删除。

这种做法的优点是节约代码量，减少编程时间。但它的缺点也是很严重的：如果插入量和删除量都很大，这种删除方式会在平衡树中留下大量的“废节点”，浪费空间，还影响效率。而且为了标记节点删除，我们还需要在节点定义中添加一个记录节点是否被删除的域。所以，只有在能够确定平衡树的吞吐量不是很大，或者不同数据的个数有限时，可以使用懒惰删除。

(2) 查找最值

在 BST 的删除中，我们需要通过找待删除节点的后继(或前驱)，也就是其右子树的最大值(左子树的最小值)。在平衡树中查找最值也是经常会用到的一种操作，方法很简单。

查找一个子树的最小值，从子树的根开始访问，如果当前节点左子节点非空，访问当前节点的左子节点；如果当前节点左子节点已经为空，那么当前节点的值就是这个子树的最小值。

同理，**查找一个子树的最大值**，从子树的根开始访问，如果当前节点右子节点非空，访问当

[9] 实用性指考虑灵活性、综合时间效率和编程难度后的综合实用性。

前节点的右子节点；如果当前节点右子节点已经为空，那么当前节点的值就是这个子树的最大值。

代码 10 为在一个给定的 Treap 中查找最大值和最小值的非递归实现代码。

```
Treap_Node *root;

int Treap_FindMin(Treap_Node *P)
{
    while (P->left) //如果有左子节点，访问左子节点
        P=P->left;
    return P->value;
}

int Treap_FindMax(Treap_Node *P)
{
    while (P->right) //如果有右子节点，访问右子节点
        P=P->right;
    return P->value;
}

int main()
{
    int Min,Max;
    Min=Treap_FindMin(root);//在 Treap 中查找最小值
    Max=Treap_FindMax(root);//在 Treap 中查找最大值
    return 0;
}
```

代码 10

(3) 前驱与后继

求一个元素在平衡树(或子树)中的**前驱**，定义为*查找该元素在平衡树中不大于该元素的最大元素*。相似的，求一个元素在平衡树(或子树)中的**后继**，定义为*查找该元素在平衡树中不小于该元素的最小元素*。

从定义中看出，求一个元素在平衡树中的前驱和后继，这个元素不一定是平衡树中的值，而且如果这个元素就是平衡树中的值，那么它的前驱与后继一定是它本身。

我们给出求前驱的基本思想：贪心逼近法。在树中查找，一旦遇到一个**不大于**这个元素的值的节点，更新当前的最优的节点，然后在当前节点的右子树中继续查找，目的是希望能找到一个更接近于这个元素的节点。如果遇到大于这个元素的值的节点，不更新最优值，在当前

节点的左子树中继续查找。直到遇到空节点，查找结束，当前最优的节点的值就是要求的前驱。求后继的方法与上述相似，只是要找**不小于**这个元素的值的节点。下面是具体的算法描述。

求前驱：

1. 从根节点开始访问，初始化最优节点为空节点；
2. 如果当前节点的值不大于要求前驱的元素的值，更新最有节点为当前节点，访问当前节点的右子节点；
3. 如果当前节点的值大于要求前驱的元素的值，访问当前节点的左子节点；
4. 如果当前节点是空节点，查找结束，最优节点就是要求的前驱。

求后继：

1. 从根节点开始访问，初始化最优节点为空节点；
2. 如果当前节点的值不小于要求前驱的元素的值，更新最有节点为当前节点，访问当前节点的左子节点；
3. 如果当前节点的值小于要求前驱的元素的值，访问当前节点的右子节点；
4. 如果当前节点是空节点，查找结束，最优节点就是要求的后继。

在求前驱和后继的过程中，我们恰恰访问了从根到叶节点的一条完整的路径。由于 Treap 的深度是 O(logN)的，所以求前驱和后继算法的时间复杂度为 O(logN)。代码 11 为在一个已知的 Treap 中求值为 5 的元素前驱和后继的代码。

```
Treap_Node *root;

//访问节点 P，查找 value 的前驱，当前最优节点为 optimal
Treap_Node * Treap_pred(Treap_Node *P,int value,Treap_Node *optimal)
{
    if (!P)
        return optimal;   //访问到空节点，返回最优节点，查找结束
    if (P->value <= value)
        return Treap_pred(P->right,value,P); //更新最优值，在右子树中继续查找
    else
        return Treap_pred(P->left,value,optimal); //左子树中继续查找
}

//访问节点 P，查找 value 的后继，当前最优节点为 optimal
Treap_Node * Treap_succ(Treap_Node *P,int value,Treap_Node *optimal)
{
    if (!P)
        return optimal;   //访问到空节点，返回最优节点，查找结束
    if (P->value >= value)
        return Treap_succ(P->left,value,P); //更新最优值，在左子树中继续查找
    else
        return Treap_succ(P->right,value,optimal); //在右子树中继续查找
```

```
}

int main()
{
    Treap_Node *pred,*succ;
    pred=Treap_pred(root,5,0); //查找 5 在 Treap 中的前驱
    succ=Treap_succ(root,5,0); //查找 5 在 Treap 中的后继
    return 0;
}
```

代码 11

根据前驱和后继的定义，我们还可以以此来查找某个元素与 Treap 中所有元素绝对值之差最小元素。如果按照数轴上的点来解释的话，就是求一个点的最近距离点。方法就是分别求出该元素的前驱和后继，比较前驱和后继哪个距离基准点最近。

求前驱、后继和距离最近点是许多算法中经常要用到的操作，Treap 都能够高效地实现。

(4) 合并重复节点

Treap 中很可能存在值相同的节点，在某些特殊情况下，重复的节点可能会有很多，但是我们却把它们分别存成一个个节点。我们有一种常数级的优化，把重复的节点合并为一个节点。方法就是在 Treap 节点中增加一个域，记录相同的这个值的个数，称为节点的**权值**，记为 weight。在插入时，如果找到了已存在的相同的值，不必再开辟新的节点，只需把已有的节点的权值增加 1。删除时，只需把权值减少 1，如果权值为 0 时，才对节点正式删除。

这种优化的效果很好，首先是在插入时节省了开辟空间的时间。更好的是在删除时，避免了大量的旋转。当重复的值非常多的时候，这种优化是十分惊人的。

代码 12 为带重复计数的 Treap 节点的定义。

```
struct Treap_Node
{
    Treap_Node *left,*right; //节点的左右子树的指针
    int value,fix,weight; //节点的值和修正值，weight 为权值
};
```

代码 12

(5) Treap 中元素的类型与排序的规则

到这里为止，上文中提到的 Treap 中元素的类型，我们都默认为了整数型。但实际上类型并

没有限制，只要是能够比较大小的类型，例如浮点数型、字符串型，也可以是复合类型(结构类型、对象类型)。

假若我们要实现一种多关键字类型排序的 Treap，我们只需自定义大于、小于、等于运算符的意义(运算符重载)，使它们有确定的大小关系。这样就可以在不修改 Treap 各种操作代码的基础上实现多关键字类型排序的 Treap。Treap 定义中“左小于中小于右”，仅仅是逻辑上的定义，实际上我们可以以任何有序的规则排序，即使是左大于中大于右，只需要在比较元素大小的函数中修改定义即可。

在以上时间复杂度的分析中，我们默认两个元素大小比较时间为 O(1)，但实际上某些复杂的类型间比较大小不是 O(1)的，如字符串，是 O(L)，L 为字符串长度。平衡树并不适合作为所有数据类型的数据的有序存储容器，因为可能有些类型的两个元素直接相互比较大小是十分耗时的，这个常数时间的消耗是无法忍受的。例如字符串，作为检索字符串的容器，我们更推荐 Trie[10]，而不是平衡树。平衡树仅适合做元素间相互比较时间很少的类型的有序存储容器。

(6) 维护子树大小的必要性

Treap 是一种排序的数据结构，如果我们想查找第 k 小的元素或者询问某个元素在 Treap 中从小到大的排名时，我们就必须知道每个子树中节点的个数。我们称以一个子树的所有节点的权值之和，为**子树的大小**。由于插入、删除、旋转等操作，会使每个子树的大小改变，所以我们必须对子树的大小进行动态的维护。

对于**旋转**，我们要在旋转后对子节点和根节点分别重新计算其子树的大小。

对于**插入**，新建立的节点的子树大小为 1。在寻找插入的位置时，每经过一个节点，都要先使以它为根的子树的大小增加 1，再递归进入子树查找。

对于**删除**，在寻找待删除节点，递归返回时要把所有的经过的节点的子树的大小减少 1。要注意的是，删除之前一定要保证待删除节点存在于 Treap 中。

代码 13 为维护子树大小的 Treap 节点的定义，以及旋转的代码。

```
struct Treap_Node
{
    Treap_Node *left,*right; //节点的左右子树的指针
    int value,fix,weight,size; //节点的值，修正值，重复计数，子树大小
    inline int lsize(){ return left ?left->size ?0; } //返回左子树的节点个数
    inline int rsize(){ return right?right->size?0; } //返回右子树的节点个数
};
```

[10] Trie 是字符串检索树，它基于对字符串中的每个字符建立节点，实现字符串的查找。对于长度为 L 的字符串，插入和查找时间都为 O(L)。空间复杂度取决于字符集大小。

```
void Treap_Left_Rotate(Treap_Node *&a) //左旋 节点指针一定要传递引用
{
    Treap_Node *b=a->right;
    a->right=b->left;
    b->left=a;
    a=b;
    b=a->left;
    b->size = b->lsize() + b->rsize() + b->weight;
    a->size = a->lsize() + a->rsize() + a->weight;
}

void Treap_Right_Rotate(Treap_Node *&a) //右旋 节点指针一定要传递引用
{
    Treap_Node *b=a->left;
    a->size = b->rsize() + a->rsize() + a->weight;
    b->size = b->lsize() + a->size + b->weight;
    a->left=b->right;
    b->right=a;
    a=b;
    b=a->left;
    b->size = b->lsize() + b->rsize() + b->weight;
    a->size = a->lsize() + a->rsize() + a->weight;
}
```

代码 13

(7) 查找排名第 k 的元素

只有当我们维护以每个节点为根的子树的大小，才能查找排名第 k 的元素[11]。根据 Treap 的一个重要性质，Treap 的子树也是 Treap，我们可以用分而治之的思想来查找排名第 k 的元素。

首先，在一个子树中，根节点的排名取决于其左子树的大小，如果根节点有权值 weight，则根节点 P 的排名是一个闭区间 A，且 A = [P.left.size + 1,P.left.size + P.weight]。根据此，我们可以知道，如果查找排名第 k 的元素，k∈A，则要查找的元素就是 P 所包含元素。如果 k<A，那么排名第 k 的元素一定在左子树中，且它还一定是左子树的排名第 k 的元素。如果 k>A，则排名第 k 的元素一定在右子树中，是右子树排名第 k-(P.left.size + P.weight)的元素。根据这种策略，我们可以总结出算法[12]：

1. 定义 P 为当前访问的节点，从根节点开始访问，查找排名第 k 的元素；

[11] 一般来说，排名第 k 的元素就是第 k 小(良序关系)的元素就是元素在 Treap 所有元素从小到大的排列中，排名第 k 的元素。注意要区别于第 k 大的元素。

[12] 该算法应用前要保证 1<=k<=root.size(root 为根节点，root.size 为整个树的大小)。

2. 若满足 P.left.size + 1 <=k <= P.left.size + P.weight，则当前节点包含的元素就是排名第 k 的元素；
3. 若满足 k < P.left.size + 1，则在左子树中查找排名第 k 的元素；
4. 若满足 k > P.left.size + P.weight，则在右子树中查找排名第 k-(P.left.size + P.weight)的元素。

代码 14 为在一个给定的 Treap 中查找排名第 8 的元素。

```
Treap_Node *root;

Treap_Node * Treap_Findkth(Treap_Node *P,int k)
{
    if (k < P.lsize() + 1)   //左子树中查找排名第 k 的元素
        return Treap_Findkth(P->left,k);
    else if (k > P.lsize() + P.weight) //在右子树中查找排名第 k-(P.lsize() + P.weight)的元素
        return Treap_Findkth(P->right,k-(P.lsize() + P.weight));
    else
        return P; //返回当前节点
}

int main()
{
    Treap_Node *result;
    result=Treap_Findkth(root,8); //在 Treap 中查找排名第 8 的元素
    return 0;
}
```

代码 14

根据上述算法，我们还可以实现查找逻辑第 k 大元素，即查找第(root.size - k + 1)小元素，root.size 为整个 Treap 的大小。甚至我们可以取代专门写的查找最值的算法。由于查找路径必定是一条子树上的路径，长度不会超过 Treap 的深度，所以时间复杂度为 O(logN)。

(8) 求元素的排名

我们通过排名可以找到对应元素，也希望求出元素在 Treap 中排名，或者成为求元素的秩。我们规定，如果在 Treap 照片那个有多个重复的元素，则这个元素的排名为最小的排名。例如 1,2,4,4,4,6 中，4 的排名为 3。在 Treap 中求元素的排名的方法与查找第 k 小的数是很相似的，可以近似认为是互为逆运算[13]。

我们的基本思想是查找要求的元素在 Treap 中的位置，且在查找路径中统计出小于要求的元素的节点的总个数，要求的元素的排名就是总个数+1。算法为：

1. 定义 P 为当前访问的节点，cur 为当前已知的比要求的元素小的元素个数。从根节点开

[13] 如果无相同的元素，求排名与求第 k 小元素互为严格的逆运算。

始查找要求的元素，初始化 cur 为 0；
2. 若要求的元素等于当前节点元素，要求的元素的排名为区间[P.left.size + cur + 1, P.left.size + cur + weight]内任意整数；
3. 若要求的元素小于当前节点元素，在左子树中查找要求的元素的排名；
4. 若要求的元素大于当前节点元素，更新 cur 为 cur + P.left.size+weight，在右子树中查找要求的元素的排名。

代码 15 为在一个已知的 Treap 中求元素 5 的排名。

```
Treap_Node *root;

int Treap_Rank(Treap_Node *P,int value,int cur) //求元素 value 的排名
{
    if (value == P->value)
        return P->lsize() + cur + 1; //返回元素的排名
    else if (value < P->value) //在左子树中查找
        return Treap_Rank(P->left,value,cur);
    else //在右子树中查找
        return Treap_Rank(P->right,value,cur + P->lsize() + weight);
}

int main()
{
    int rank;
    rank=Treap_Rank(root,8,0); //在 Treap 中求元素 8 的排名
    return 0;
}
```

代码 15

(9) 维护附加关键字

有时候，我们建立 Treap 维护的顺序关键字并不是我们主要关心的内容，而要关心的是附加关键字。根据不同目的维护的附加关键字的处理方法也不尽相同，下文仅仅以一个例子介绍附加关键字的处理方法。

【例题 1】
顺序前缀和[14]

[问题描述]
要求维护一个由二元组构成的序列，使序列中每个元素按第一关键字升序排列。操作包括：添加一个新元素，删除一个已有元素，查询这个序列的第二关键字最大前缀和。

[14]题目来源：BYVoid 原创。

例如已知的序列{(1,0), (3,-2), (4,-3), (6,3), (7,-1)}有如下几项操作：添加(3,1)入序列，添加(1,1)入序列，从序列中删除(4,-3)，查询最大前缀和。第 1 次操作后，序列变成了{(1,0), (3,1), (3,-2), (4,-3), (6,3), (7,-1)}，第 2 次操作后，序列变成了{(1,1), (1,0), (3,1), (3,-2), (4,-3), (6,3), (7,-1)}，第 3 次操作后，序列变成了{(1,1), (1,0), (3,1), (3,-2), (6,3), (7,-1)}。此时序列的 i 项前缀的和 S[i]={1, 1, 2, 0, 3, 2}，所以序列的最大前缀和是前 5 项和，值为 3。

解析

由于序列总是要求升序排列，我们可以想到以元素第一关键字升序排列，使用 Treap 维护。由于每次要查询的是第二关键字构成的序列的最大前缀和，我们可以容易想到，对于第一关键字相同的元素，第二关键字大的元素应放在前面。

规定排序的顺序之后，我们要考虑如何维护前 i 项元素的第二关键字的和(以下简称前 i 项的和)。设每个节点的第一关键字为 a，第二关键字为 b，我们要在 Treap 中的节点添加附加域 sum，表示以该节点为根的子树中所有元素的第二关键字和，以及附加域 max，表示以该节点为根的子树中所有元素构成的顺序序列最大的前缀和。

sum 值可以像维护子树的大小 size 值一样的递归地维护，而且旋转时也要重新计算。而对于节点 p 的 max 值则要分情况讨论：
情况一，当前子树最大前缀的结尾在该节点的左子树，此时 p.max = p.left.max；
情况二，当前子树最大前缀的结尾恰好是该节点，此时 p.max = p.left.sum + p.b；
情况三，当前子树最大前缀的结尾在该节点的右子树，p.max = p.left.s + p.b + p.right.max。

在实际的插入和删除过程中，每次旋转后都要重新计算 sum 值，然后依次计算旋转后的子节点和根节点的 max 值。维护好后，每次查询最大前缀和，只需要 O(1)的时间。

时可要记住 Treap 是一种二叉树结构，它具有良好的分治结构，所以在维护各种具体的附加关键字时，二分或三分递推的思想一般都是解决问题的关键。

四、 Treap 的应用

了解 Treap 这种平衡树的数据结构以后，要付之于应用中。本章讨论的内容为 Treap 的各种实际应用[15]。

1. Treap 在动态统计问题中的应用

Treap 是一种高效的动态的数据容器，据此我们可以用它处理一些数据的动态统计问题。本节内容以两个例题介绍了对于直观的动态统计问题的一般解决方法。

【例题 2】

[15]本章大多内容均也可以用其他的平衡树实现。

排名系统[16]

[问题描述]
有一个游戏排名系统，通常要应付三种请求：上传一条新的得分记录、查询某个玩家的当前排名以及返回某个区段内的排名记录。当某个玩家上传自己最新的得分记录时，他原有的得分记录会被删除。为了减轻服务器负担，在返回某个区段内的排名记录时，最多返回 10 条记录。

[输入]
第一行是一个整数 N（10<=N<=250000）表示请求总数目。接下来 n 行,每行包含了一个请求。请求的具体格式如下：
+Name Score 上传最新得分记录。Name 表示玩家名字，由大写英文字母组成，不超过 10 个字符。Score 为最多 8 位的正整数。
?Name 查询玩家排名。该玩家的得分记录必定已经在前面上传。如果两个玩家的得分相同，则先得到该得分的玩家排在前面。
?Index 返回自第 Index 名开始的最多 10 名玩家名字。Index 必定合法，即不小于 1，也不大于当前有记录的玩家总数。

[输出]
对于?Name 格式的请求，应输出一个整数表示该玩家当前的排名。
对于?Index 格式的请求，应在一行中依次输出从第 Index 名开始的最多 10 名玩家姓名，用一个空格分隔。

解析
这是一个平衡树经典应用类型的题目。因为作为字符串的姓名是不便于处理的，我们给每个玩家都制定一个 ID，首先要建立一个由姓名到玩家 ID 的映射数据结构。为了查找快速，我们可以用字符串类型的 Treap 存储玩家姓名，但是更好的方法是用 Trie。之后我们建立一个双关键字的 Treap，关键字 1 为得分从小到大，关键字 2 为时间戳从大到小，这种排列方式的逆序，恰好是我们要的顺序(也可以直接就是逆序)。

对于+Name Score 这样的请求，首先查询 Name 玩家是否已经存在，如果已经存在，在 Treap 中删除对应已经存在的记录。然后把新的记录插入 Treap。

对于?Name 的请求，就是基本的求排名操作，但是注意从大到小的排名等于(总记录数 - 求得的排名 + 1)。

对于?Index 操作，就是分别查找第(总记录数 + 1 – k)小的记录，其中 k 满足 Index<=k<=Index + 9 且 1<=k<=总记录数。

【例题 3】
郁闷的出纳员[17]

[16]题目来源：河南 2008 年省选加试题目。
[17] 题目来源：NOI 2004 Day1。

[问题描述]
OIER 公司是一家大型专业化软件公司，有着数以万计的员工。作为一名出纳员，我的任务之一便是统计每位员工的工资。这本来是一份不错的工作，但是令人郁闷的是，我们的老板反复无常，经常调整员工的工资。如果他心情好，就可能把每位员工的工资加上一个相同的量。反之，如果心情不好，就可能把他们的工资扣除一个相同的量。我真不知道除了调工资他还做什么其它事情。
工资的频繁调整很让员工反感，尤其是集体扣除工资的时候，一旦某位员工发现自己的工资已经低于了合同规定的工资下界，他就会立刻气愤地离开公司，并且再也不会回来了。每位员工的工资下界都是统一规定的。每当一个人离开公司，我就要从电脑中把他的工资档案删去，同样，每当公司招聘了一位新员工，我就得为他新建一个工资档案。
老板经常到我这边来询问工资情况，他并不问具体某位员工的工资情况，而是问现在工资第 k 多的员工拿多少工资。每当这时，我就不得不对数万个员工进行一次漫长的排序，然后告诉他答案。
好了，现在你已经对我的工作了解不少了。正如你猜的那样，我想请你编一个工资统计程序。怎么样，不是很困难吧？

[输入文件]
第一行有两个非负整数 n 和 min。n 表示下面有多少条命令，min 表示工资下界。
接下来的 n 行，每行表示一条命令。命令可以是以下四种之一：

名称	格式	作用
I 命令	I_k	新建一个工资档案，初始工资为 k。如果某员工的初始工资低于工资下界，他将立刻离开公司。
A 命令	A_k	把每位员工的工资加上 k
S 命令	S_k	把每位员工的工资扣除 k
F 命令	F_k	查询第 k 多的工资

_（下划线）表示一个空格，I 命令、A 命令、S 命令中的 k 是一个非负整数，F 命令中的 k 是一个正整数。
在初始时，可以认为公司里一个员工也没有。

[输出文件]
输出文件的行数为 F 命令的条数加一。
对于每条 F 命令，你的程序要输出一行，仅包含一个整数，为当前工资第 k 多的员工所拿的工资数，如果 k 大于目前员工的数目，则输出-1。
输出文件的最后一行包含一个整数，为离开公司的员工的总数。

解析
与一般的修改不同，这道题要求对所有人修改，如果一个一个进行的话，修改工资的时间复杂度高达 O(N)。如果我们反过来考虑，定义一个“基准值”，把所有人的工资看作“相对工资”，就是相对于基准值。这样每次修改所有人工资仅仅需要修改基准值就行了。于是变成了一个动态统计问题，建立一个 Treap，存储相对工资。

为了方便考虑，定义基准值为 delta，相对工资 V 对应的实际工资为 F[V]，则有 F[V]=V+delta，V=F[V]-delta。定义工资下限为 lowbound 这是一个实际的下限，存储相对下限就是 lowbound-delta。

对于 I_k 插入一个新的工资记录值 k，k 为实际工资，对应的相对工资为 k-delta，应把 k-delta 插入 Treap。

对于 A_k，将基准值 delta 增加 k。对于 S_k，将基准值 delta 减少 k，然后在 Treap 中删除所有小于(lowbound-delta)的元素。

由于我们总是查询第 k 多的工资，我们可以依照例 1 的方法，求(总数-k+1)小的工资。我们也不妨换种思路，把 Treap 建立成一个关键字反序大小比较的 Treap，即在比较函数中规定如果 a>b(实际的)，则 a 小于 b(逻辑的)，a 放在 b 的左子树。这虽然难以理解，但却能够满足一定的逻辑顺序。这样建立出的 Treap 就是自然的从大到小排序的了，查询第 k 多的工资，就是求排名第 k 的元素。

相对于上一题，这道题的统计方法就不比直观，而大多数情况下都是这样的，我们需要对题充分地分析，才能正确的设计出算法和数据结构。

2. Treap 在搜索问题中的应用

搜索是人工智能实现的基本方法，其主要原理是应用了产生式系统。在广度优先搜索(Breadth First Search)中，大量的状态被产生，往往为了避免重复搜索需要使用状态判重。在此时 Treap 可以作为一个状态容器，从而实现高效的判重操作，大大提高搜索效率。

3. Treap 在动态规划问题中的应用

在动态规划中使用 Treap，一般是为了动态维护避免重复计算，或者用决策单调性来优化的一种手段。

【例题 4】
金矿[18]

[问题描述]
金矿的老师傅年底要退休了。经理为了奖赏他的尽职尽责的工作，决定在一块包含 n(n ≤ 15000) 个采金点的长方形土地中划出一块长度为 S ，宽度为 W 的区域奖励给他(1 ≤ s , w ≤ 10 000)。老师傅可以自己选择这块地的位置，显然其中包含的采金点越多越好。你的任务就是计算最多能得到多少个采金点。如果一个采金点的位置在长方形的边上，它也应当被计算在内。

[输入格式]
输入文件的第一行有两个整数,中间用一个空格隔开，表示长方形土地的长和宽即 s 和 w(1<=s,w<=10 000)。第二行有一个整数 n(1<=n<=15 000)，表示金矿数量。下面的 n 行与金矿相对应，每行两个整数 x 和 y (-30 000<=x,y<=30 000),中间用一个空格隔开，表示金矿的坐

[18] 题目来源：POI 2001 Stage 3。

标。

[输出格式]
输出文件只有一个整数，表示选择的最大金矿的数。

解析
发现坐标的范围很大，而点并不是很多，首先想到了离散化的方法。然后横向扫描每个带状的区间，对于每个带状区间，再纵向扫描其中点的个数。这种方法是最容易想到的，但是时间复杂度却高达 O(N^2logN)。关键在于优化每次确定带状区间的纵向扫描。

显然对于确定的带状区间，我们扫描时只需考虑它们的纵坐标。如果我们把一个纵坐标为 y 的点描述成两个事件(y,1)和(y+h+1,-1)，然后把所有事件按照第一关键字排序，定义 S[i]为前 i 项事件的第二关键字之和，那么 Max{S[i]}就是确定的带状区间内高度为 w 的矩形最多覆盖到的点数。为什么是这样，我们可以想象有两个挡板，分别是 y=A-h-1(下板)和 y=A(上板)，从下向上移动挡板，从而确定一个矩形。某个点的第一个事件为(y,1)，当上板 A=y 时，该点被覆盖，若 A 继续增大，使得 A-h-1=y，该点就恰好离开了覆盖区域，反过来我们可以以为是 A=y+h+1，所以定义第二个事件为(y+h+1,-1)。这样，前 i 个事件的第二关键字和 S[i]，就代表了矩形在某个位置时覆盖的点数。

带状区间的左右扫描，也可以考虑成两个挡板之间的问题，首先确定挡板的初始位置在最左端，依次向右移动左右两个挡板。确定一个区间后，统计点事件的最大前缀和成了最关键的问题。由于左右挡板是移动的，我们需要动态统计一个排序结构。于是我们想到了用平衡树，维护每个点事件的第一关键字(y 坐标)升序，并记录权。移动右挡板时，把右挡边新位置所在线上的所有点对应的两个点事件加入平衡树；移动左挡板时，把左挡板时，把左挡边所在线上的所有点事件从平衡树中删除。

为了快速统计出最大的前缀和，在此基础上，我们还需要对于平衡树中每个节点维护其子树所有节点权值和，以及子树中最大的前缀和。定义 p 的权值为 p.w，以节点 p 为根的子树的权值和为 p.s，以节点 p 为根的子树的最大的前缀和为 p.m，我们可以用树形的动态规划算出 p.m 的值。

```
p.m=Max{
    p.left.m; //最大前缀和的尾元素在左子树
    p.left.s+p.w; //最大前缀和的尾元素就是 p
    p.left.s+p.w+p.right.m; //最大前缀和的尾元素在右子树
}
```

于是全部序列的最大前缀和就是根节点 root.m。

于是得到的算法如下：

1. 建立离散化坐标系;
2. 左右扫描，确定带状区间，维护平衡树中节点;
3. 统计对于确定带状区间，平衡树中元素最大的前缀和;

算法总的时间复杂度为 O(NlogN)，可以解决问题了。上述方法巧妙得把矿点拆分成了“点事件”，利用动态规划的方法维护了点事件的最大前缀和，是解决问题开阔思路的优良途径。

4. Treap 与其他数据结构的相关应用

平衡树之所以称之为强大，不仅是因为它本身的操作灵活高效，而且是因为它还能够实现许多衍生的数据结构。在这一节我们介绍平衡树的两种衍生数据结构优先队列和集合。

(1) 优先队列的实现

优先队列(Priority Queue)是一种按优先级维护进出顺序的数据容器结构，可以选择维护实现取出最小值或最大值。我们通常用**堆**实现优先队列，通常取出最值的时间复杂度为 O(logN)。

用最小堆可以实现最小优先队列，用最大堆可以实现最大优先队列。但是如果我们要求一种“双端优先队列”，即要求同时支持插入、取出最大值、取出最小值的操作，用一个单纯的堆就不能高效地实现了。一般的解决方案是分别建立一个最大堆和一个最小堆，并且两个堆之间的元素都互相指向。插入元素时，在两个堆中分别都插入；删除最大值时，取出并删除最大堆中的最大值，更新最大堆，然后找到该元素在最小堆中的映射位置，同样删除；删除最小值时同样要分别删除两个堆中的最小值。时间都也是 O(logN)。

上述实现双端优先队列方法较为复杂，需要分别写出代码。而我们可以方便地使用 Treap 实现双端优先队列，只需建立一个 Treap，分别写出取最大值和最小值的功能代码就可以了，无需做任何修改。由于 Treap 平衡性不如堆完美，但期望时间仍是 O(logN)。更重要的是在实现的复杂程度上大大下降，而且便于其他操作的推广。所以，用 Treap 实现优先队列不失为一种便捷而又灵活的方法。

【例题 5】
促销活动[19]
[问题描述]
促销活动遵守以下规则：

1. 一个消费者 —— 想参加促销活动的消费者，在账单下记下他自己所付的费用，他个人的详细情况，然后将账单放入一个特殊的投票箱。
2. 当每天促销活动结束时，从投票箱中抽出两张账单：
   - 第一张被抽出的账单是金额最大的账单
   - 然后被抽出的是金额最小的账单，对于付了金额最大账单的这位消费者，将得到一定数目的奖金，其奖金数等于他账单上的金额与选出的最小金额的差。
   - 为了避免一个消费者多次获奖，根据上面所抽出的两张账单都不返回到投票箱，但是剩下的账单还继续参加下一天的促销活动。

超市的售出额是巨大的，这样可以假定，在每天结束，拿出数额最大账单和数额最小账之前，在投票箱内就已经至少存在了 2 张账单。你的任务是去计算每天促销活动投进投票箱的账

[19] 题目来源：POI 2000 Stage 3

单数额的基本信息。在整个活动中开销总数。
本题中约定：
整个活动持续了 N 天 (N<=5000) 。 第 i 天放入的帐单有 a[i] 张， a[i]<=10^5 。且 Sum(a[1]...a[n])<=10^6 。 每一天放入的帐单的面值均 <=10^6 。

[输入格式]
输入文件的第一行是一个整数 n （ 1 <= n <= 5000 ），表示促销活动历时的天数。
以下的 n 行，每行包含若干由空格分隔的非负整数。第 i+1 行的数表示在第 i 天投入箱子的账单金额。每行的第一个数是一个整数 k （ 0 <= k <= 10^5 ）， 表示当日账单的数目。后面的 k 个正整数代表这 k 笔账单的金额，均小于 10^6 。
整个活动中涉及到的账单笔数不会超过 10^6 。

[输出格式]
输出文件的唯一一行是一个整数，等于整个促销活动中应该付出的奖金总额。

解析
由题意可知，“投票箱”实际就是一个双端优先队列。我们只需建立一个 Treap，维护所有插入和取出最值的操作，就能解决问题。

(2) 数据结构的复合——树套树

在各种树形的数据结构中，经常会出现多种结构复合使用的例子，俗称树套树。树套树可以是平衡树、线段树、并差集、Trie 树、后缀树等。线段树套平衡树的是最常见的一种复合方式。

【例题 6】
动态排名系统[20]

[问题描述]
给定一个长度为 N 的已知序列 A[i](1<=i<=N)，要求维护这个序列，能够支持以下两种操作：
1、查询 A[i],A[i+1],A[i+2],...,A[j](1<=i<=j<=N)中，升序排列后排名第 k 的数。
2、修改 A[i]的值为 j。

所谓排名第 k，指一些数按照升序排列后，第 k 位的数。例如序列{6,1,9,6,6}，排名第 3 的数是 6，排名第 5 的数是 9。

[输入格式]
第一行包含一个整数 D(0<=D<=4)，表示测试数据的数目。接下来有 D 组测试数据，每组测试数据中，首先是两个整数 N(1<=N<=50000),M(1<=M<=10000)，表示序列的长度为 N，有 M 个操作。接下来的 N 个不大于 1,000,000,000 正整数，第 i 个表示序列 A[i]的初始值。然后的 M 行，每行为一个操作

[20] 题目来源：zoj (zju online judge) 2112 转述。

Q i j k 或者
C i j
分别表示查询 A[i],A[i+1],A[i+2],...,A[j](1<=i<=j<=N)中，升序排列后排名第 k 的数，和修改 A[i]的值为 j。

[输出格式]
对于每个查询，输出一行整数，为查询的结果。测试数据之间不应有空行。

解析
解决这类区间问题，常常要用到线段树，而动态排名的查询又要用到平衡树，所以建立一棵线段树，维护 N 的节点的所有区间，然后在每个线段树的节点上建立一个平衡树，用来维护当前区间内所有数的动态排名。很显然，每个线段树节点上的平衡树，都是这个线段树节点的两个子节点的平衡树的合并。由于我们无法实现高效的平衡树合并，只好用这种以空间换时间的方法。

对于每个修改操作，只需先在线段树找出单位区间[i,i]上平衡树中的唯一节点，就是 A[i]的原值，然后把线段树从根到该节点路径上所有节点的平衡树中都删除掉 A[i]的原值，然后插入新值 j。

查询操作为比较特殊，因为给定的区间[i,j]可能对应线段树中若干个区间的并，所以首先找出这对应的 q 区间的平衡树 T[1],T[2],T[3],...,T[q]，找出其中的最大值 Max 和最小值 Min，然后在[Min,Max]之间二分答案。对于给定的答案 r，判断是符合要求，否则按照单调性缩小二分区间。

如何求给定的 r 值的总排名，由于 r 可能不在某些甚至所有的 q 个平衡树中，所以有时求 r 的排名是没有意义的，换而我们可以求 r 在所有平衡树中后继的最小值 MinSucc 的排名。对于第 i 个平衡树，要求 r 在 T[i]中的后继(不大于 r 的最小值)的排名 S[i]，MinSucc 的排名就是 k1= Σ{S[i]-1} + 1。但是相同的 MinSucc 的值可能会有多个，所以 MinSucc 的排名实际上是属于一个区间的，应在所有的平衡树中找出 MinSucc 一共的个数 Count，MinSucc 的最大排名就是 k2=k1+Count-1。如果 k 属于[k1,k2]，那么结果就是 MinSucc。如果 k2<k，则应向增大的方向二分答案，否则向减小的方向二分答案。

由于在线段树、平衡树和二分答案时要三次二分，所以这个算法的时间复杂度为 $O(N(\log N)^3)$。

五、总结

Treap 作为一种简洁高效的有序数据结构，在计算机科学和技术应用中有着重要的地位。它可以用来实现集合、多重集合、字典等容器型数据结构，也可以用来设计动态统计数据结构。

六、参考资料

1. 中文维基百科 http://zh.wikipedia.org/wiki/排序算法

2. 英文维基百科 http://en.wikipedia.org/wiki/Sorting_algorithm
3. [CLRS 笔记 12，二叉查找树/红黑树] http://hideto.javaeye.com/blog/286943
4. [二分法与统计问题] 淮阴中学 李睿
5. [Randomized Search Trees] Raimund Seidel
6. [Treap 和 Skip List 的总结] LogicalMars
7. [怀疑中的平衡树] 小鱼兒
8. [DST 详解] 风过叶落
9. [神奇的 Splay The Magical Splay] sqybi
10. [Size Balanced Tree] Chen Qifeng (Farmer John)