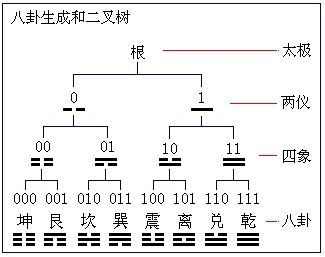
### 二叉树及其应用

“易有太极，是生两仪，两仪生四象，四象生八卦。”——《易传·系辞上传》。

大家一定都听听说过上面这句话，但是你知道吗，这句话可以用图2.1的树形式更加直观的表示出来。



* **图2.1 八卦图**

可以观察到，图2.1不是一棵普通的树，它有着非常优美的结构。这就是接下来要讨论的内容——二叉树。

**§**2.1 二叉树的概念及其性质

**一、 二叉树的概念**

什么是二叉树？顾名思义，它的每个非叶子节点最多只能有两个孩子的树结构。为了方便研究，我们对节点的两个孩子进行命名和编号，将左边的节点成为左孩子（left child，简写为lchild），右边的节点称为右孩子（right child，简写为rchild），那么以孩子为根的子树分别称为“左子树”（left subtree，简写为ltree）和“右子树”（right subtree，简写为rtree）。如图2.2

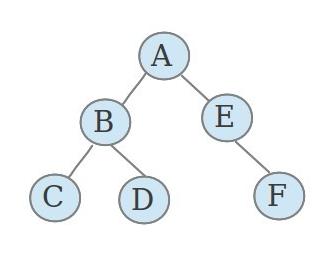


图2.2二叉树

一般情况下左右孩子有序，不能相互颠倒。

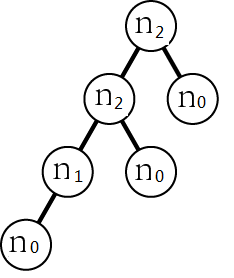
**二、二叉树的性质**

由于二叉树结构的优美性，因此具有很好的性质。

**性质1：**若二叉树的叶子数为n0，度为2的节点数为n2，则n0=n2+1。

这个性质表明在二叉树中，叶子节点的个数与度为1的节点个数无关。

设度为0的节点数为n0，度为1的节点数为n1，度为2的节点数为n2，图2.3中n=6, e=5，n0=3, n1=1, n2=2,有n0=3=2+1=n2+1**。**



**图2.3 一棵普通的二叉树**

证明：

显然n=n0+n1+n2 (树上只有这三种点)，n=e+1 (n个点的树有n-1条边)

∵ e=n1+2\*n2 (每个n1点向下有1条边，每个n2点向下有2条边)

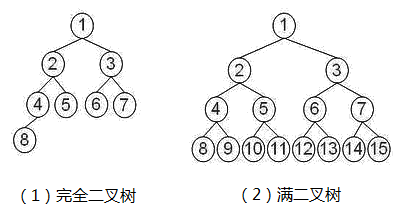
∴ n=n0+n1+n2=e+1=n1+2\*n2+1

∴ n0=n2+1

**性质2：**深度为k的二叉树，最多只有2k-1节点。

证明：树的第一层只有1个点，第二层最多2个点，第三层最多4个点，...第k层最多2k-1个节点。所以，二叉树最多节点数=20+21+22+...+2k-1=2k-1。

特别的，深度为k，且有2k-1个节点的二叉树，称为**满二叉树**（如图2.3）。这种树的特点是每一层上的节点数都是最大节点数。而在一棵二叉树中，除最后一层外，其余层都是满的，并且最后一层或者是满的，或者是在**右边缺少连续若干节点**，此二叉树称为**完全二叉树**。显然，深度为k的完全二叉树，至少有2k-1个节点，至多有2k-1个节点。



**图2.4 完全二叉树示例**

**性质3：**具有n个节点的完全二叉树的深度为log2n+1

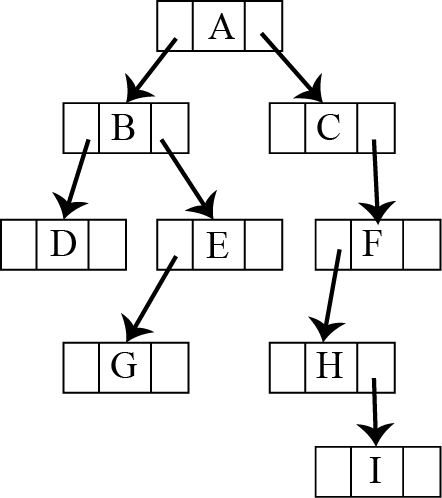
**性质4：**若将完全二叉树的每个节点从上至下，从左至右进行编号，那么，对标号为x的节点，若x存在左孩子，则左孩子的标号为2x，若x存在右孩子，则右孩子的标号为2x+1。反之，若x有父亲，则父亲的编号为x/2取整。

（证明略）

**§2.2 二叉树的存储方法**

1. **二叉树的孩子表示法**

对每一个节点，存储该节点的左右孩子。如图2.5



**图2.5 二叉树的孩子表示法**

存储结构代码：

struct node { // 存储关于这个节点的一些信息

int lc; //左孩子节点编号

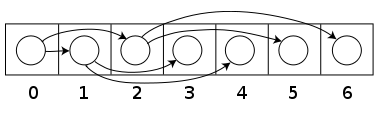
int rc; //右孩子节点编号

node() {lc = rc = 0;} // 初始化左右孩子都不存在，都是0

};

1. **二叉树的数组表示法**

**图2.6**



构建代码：

int ch[N][2]; // ch[x][0]表示x的左孩子，ch[x][1]表示x的右孩子

**三、完全二叉树的数组表示法**

根据之前所讲的完全二叉树的性质，只要确定了节点个数n，完全二叉树的形态也确定了，而每个对于节点x，它的左儿子是2x，右儿子是2x+1，父亲是x/2。因此无需存储任何和树的形态有关的信息。只需要用一个一维数组来表示每个节点上的信息。这种优美的存储方法在接下来“二叉堆”的内容中将非常有用。

**§2.3二叉树的遍历**

前序、中序、后序遍历是树的三种遍历方式。他们之间的区别，在于根的在遍历中的出现顺序。

前序遍历：根——左——右，图2.2的遍历前序结果是ABCDEF

中序遍历：左——根——右，图2.2的中序遍历结果是CBDAEF

后序遍历：左——右——根，图2.2的后序遍历结果是CDBFEA

**例2.1**给出一棵二叉树，求它的前序遍历，中序遍历，后序遍历。

前序遍历：

void preorder\_traversal(int x) {

printf("%d ", x);

if (ch[x][0]) preorder\_traversal(ch[x][0]);

if (ch[x][1]) preorder\_traversal(ch[x][1]);

}

中序遍历：

void inorder\_traversal(int x) {

if (ch[x][0]) inorder\_traversal(ch[x][0]);

printf("%d ", x);

if (ch[x][1]) inorder\_traversal(ch[x][1]);

}

后序遍历：

void postorder\_traversal(int x) {

if (ch[x][0]) postorder\_traversal(ch[x][0]);

if (ch[x][1]) postorder\_traversal(ch[x][1]);

printf("%d ", x);

}

**例2.2** 给出一棵树的前序遍历序列和中序遍历序列，求出这棵二叉树。

图2.7给出的前序遍历序列和中序遍历序列分别为GDAFEMHZ和ADEFGHMZ

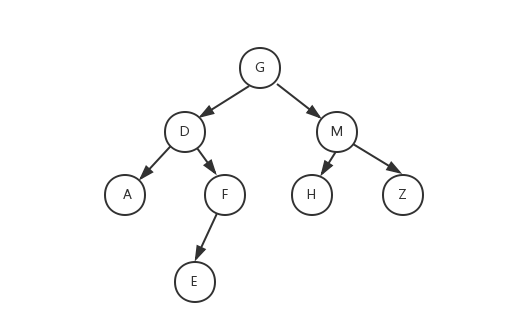


图2.7根据前序和中序遍历得到的二叉树

分析：可以这样考虑前序遍历和中序遍历：

前序遍历：root，{左子树的节点}，{右子树的节点}

中序遍历: {左子树的节点}，root，{右子树的节点}

那么可以每次通过root在中序遍历中的位置，确定出左子树节点的集合和右子树的节点的集合，而左子树和右子树也分别是一棵二叉树，所以得到了两个和原问题相同的子问题，可以递归求解，直到只剩下一个节点。

代码如下：

int solve(int ls, int rs, int lt, int rt) { // 将前序遍历和中序遍历分别存在S和T中

if (ls > rs) return 0; // 这是一个空节点

if (ls == rs) {

printf("For %c, it is a leaf\n", S[ls]); // 输出它是一个叶子

return ls; // 递归的最底层

}

int pos;

for (int i = lt; i <= rt; i++) if (T[i] == S[ls]) {pos = i; break;}

int len = pos - lt; // 左子树的大小

int lc = solve(ls + 1, ls + len, lt, pos - 1); // 递归左子树

int rc = solve(ls + len + 1, rs, pos + 1, rt); // 递归右子树

printf("For %c, left child is %c, right child is %c\n", S[ls], S[lc], S[rc]); // 分别输出左儿子和右儿子

return ls;

}

int main() {

scanf("%s", S + 1); n = strlen(S + 1); S[0] = '#';

scanf("%s", T + 1);

solve(1, n, 1, n);

return 0;

}

**例2.3** 给定一棵二叉树的前序遍历序列和后序遍历序列，求有多少种可能的二叉树，保证至少存在一种答案。由于答案可能很大，输出答案除以109+7的余数即可。

由例2.2可知，已知二叉树的前序遍历序列和中序遍历序列可以唯一确定该二叉树。因此该题也可转化为求有多少种可能的中序遍历序列。

首先考虑前序遍历的第一个元素x，显然它是树的根。因此在后序遍历中x一定是最后一个数。于是可以去掉这两个序列的x。如果此时序列为空，那么已经做完，否则余下的前序遍历中的第一个数y则是x的左/右子树的根。在后序遍历中找到y的位置，如果y此时是最后一个，意味着x只有一棵子树，但无法确定y是左子树还是右子树，也就意味着是左右都满足条件，于是将答案乘2，继续递归计算。否则，可以确定y是x的左子树，同时也知道了x的左右子树分别在哪个区间，于是分成两部分递归计算再将答案相乘就可以了。时间复杂度O(n)。

代码如下：

#include<cstdio>

const int mod=1e9+7;//模数

const int N=1e5+5;

int n;

int a[N],b[N];

int posa[N],posb[N];//这里存每个数字在数组a/b中出现的位置

int solve(int al,int ar,int bl,int br){ //求a[al..ar]与b[bl..br]对应的答案

if(al==ar) return 1;//基础情况

int y=a[al+1];

if(posb[y]==br-1)

return (long long) solve(al+1,ar,bl,br-1)\*2%mod;//左右子树均有可能答案乘2

else

return (long long)solve(al+1,al+posb[y]-bl+1,bl,posb[y])\*

solve(al+posb[y]-bl+2,ar,posb[y]+1,br-1)%mod;

//左/右子树确定，左右子树答案相乘。注意中间结果用long long存储以免溢出。

}

signed main()

{

register int i;

scanf("%d",&n);

for(i=1;i<=n;i++) scanf("%d",&a[i]);

for(i=1;i<=n;i++) scanf("%d",&b[i]);

//预处理

for(i=1;i<=n;i++) posa[a[i]]=i;

for(i=1;i<=n;i++) posb[b[i]]=i;

printf("%d\n",solve(1,n,1,n));

return 0;

}

**§2.4 树、森林与二叉树的转化**

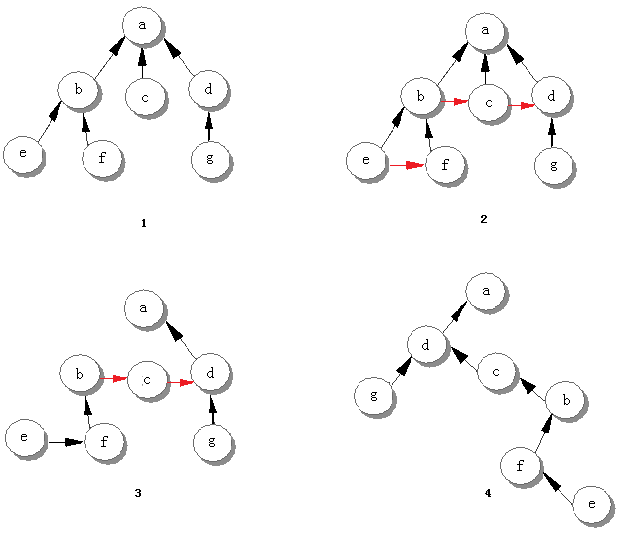
1. **孩子兄弟表示法**

在处理一些多叉树的问题时，常常因为树的分支过多而不好处理。

事实上，可以采用“左儿子右兄弟”的方法，将一棵多叉树转为二叉树，方便处理一般的多叉树问题。

将一棵多叉树（称之为原树）转为二叉树（称之为新树）。对于某个节点，把其在原树上的第一个儿子节点作为在新树上的左儿子，把原树中它的下一个兄弟作为它的右儿子。

这样，可以直接在原树上进行遍历，来实现二叉树的转换过程。如图2.8



**图2.8 树与二叉树的转化示意图**

1.原树（边读入的顺序是从左向右，因为采用邻接表，所以最靠右的儿子节点是第一个儿子节点）

2.由儿子节点向前一个儿子节点连边

3.去掉原树中的边，保留第一个儿子节点连向父亲节点的边

4.原树中的边（黑边）为连向左孩子的边，新边作为连向右孩子的边

代码实现（原树用邻接表存储）：

int u[MAXN], v[MAXN \* 2], p[MAXN \* 2], m, f[MAXN], now;

int ch[MAXN][2];

void ss(int x) {

if ( u[x] == 0 ) return;

ch[x][0]=v[u[x]];

ss( v[u[x]] );

int pre=v[u[x]];

for ( int i=p[u[x]]; i; i=p[i] ) {

ch[pre][1]=v[i];

ss( v[i] );

pre=v[i];

}

}

**二、森林转化为二叉树**

事实上，可以注意到一棵树转化为二叉树以后，根节点一定没有右儿子——这是因为根节点没有兄弟。这体现出树转二叉树的方法似乎可以进一步扩展到根有右儿子的情况。不难发现还可以将森林也转化为二叉树。具体的，新建一个虚节点，让森林中的所有树的根依次成为它的孩子，这样就得到了一棵新的树。对这棵树进行树转二叉树，可以发现虚节点成为了一个只有左儿子的根节点，那么，此时将虚节点去掉，剩下的依然是一棵二叉树，并且新的根节点也可以拥有右儿子了。这使得二叉树的形态更加优美。

**三、二叉树转化为树/森林**

转化回去的过程也并不困难，只需要牢记住：二叉树上的左儿子是它的第一个儿子，右儿子是它的兄弟即可。

**四、树转二叉树的实例**

**例2.4** 给定一棵有根树，对于每个非叶节点，给定其每个儿子节点的标号顺序。要求对于每个非根节点，若它是其父节点的第i个儿子，输出其父亲第1到第i个儿子节点的子树的大小之和。

**解法：**如果求出每个子树的大小，对于每个非根节点，暴力枚举之前的几个儿子，这样的复杂度显然不优。可以考虑采用多叉树转二叉树，这样对于每个节点的右子树，都是它后面的兄弟节点，这样，只要总节点减去右孩子节点即可。

求二叉树的大小代码：

int u[MAXN], v[MAXN \* 2], p[MAXN \* 2], m, f[MAXN], now;

int ch[MAXN][2];

void getsize(int x) {

size[x]=1;

if(ch[x][0]){ getsize(ch[x][0]); size[x]+=size[ch[x][0]]; }

if(ch[x][1]){ getsize(ch[x][1]); size[x]+=size[ch[x][1]]; }

}

**§2.5哈夫曼树及其应用**

1. **哈夫曼树的概念**

一棵具有n个带权叶节点的二叉树，使得所有叶节点的带权路径长度（叶节点\*叶节点到根节点的路径长度）之和最小，这样的二叉树被称为最优二叉树，也称哈夫曼树（Huffman Tree）。

一个简单性质：哈夫曼树的每个非叶节点都有两个儿子，否则不优。

1. **哈夫曼树的构建**
2. 将n个带权节点，作为n棵只有一个节点的树。
3. 选择两棵根节点权值最小的树，将这两棵树分别作为左右儿子合并成一颗树，并将根节点的权值赋为左右儿子的权值之和。
4. 重复执行第2步n-1次，直到所有点都在一棵树中，则这棵树就是哈夫曼树。

上述构建方法，满足每个非叶节点有两个儿子节点。每次合并操作，其意义是让两个树中的叶节点的深度增加1，总权值也相应增加一次。因为每次选择的权值是最小的，所有每次增加的权值也是最小的，所有总带权路径长度之和是最小的。

建议增加构建哈弗曼树的代码

1. **哈夫曼编码**

哈夫曼编码是哈夫曼树的重要应用之一。

在数据通信中，需要将传送的信息转换成二进制编码，用0、1的不同排列来表示。比如，某篇文章有a,b,c,d,e五种字符，若用等长的二进制编码表示这五个字符，需要至少用长度为3的编码（000——a，001——b，010——c，011——d，100——e）。

对于任何一个有效的编码，它不能是另一个编码的前缀，否则无法正确识别。例如01表示a，则不能用011表示b。因为011可能会分解为01和1两个编码，从而引起歧义。

也就是说，如果把所有的二进制编码构成一棵二叉树，将左儿子设置为0，右儿子设置为1，则第L层的每个节点对应一个长度为L-1的二进制编码，显然叶子的编码不是任何一个编码的前缀，因此所有叶子节点均为有效编码。

若考虑到一篇文章的的字符频率，要使得每个字符的带权路径长度之和最小，则可以以每个字符作为叶节点的构造出一棵哈夫曼树，叶节点的编码即为这篇文章的哈夫曼编码。

一棵有5个带权值的字符A,B,C,D,E，它们的权值分别为15,7,6,6和5。构建一棵哈夫曼树过程如图2.9

a）将每个节点按权值从大到小排序。每次选出现在权值最小的两个节点。

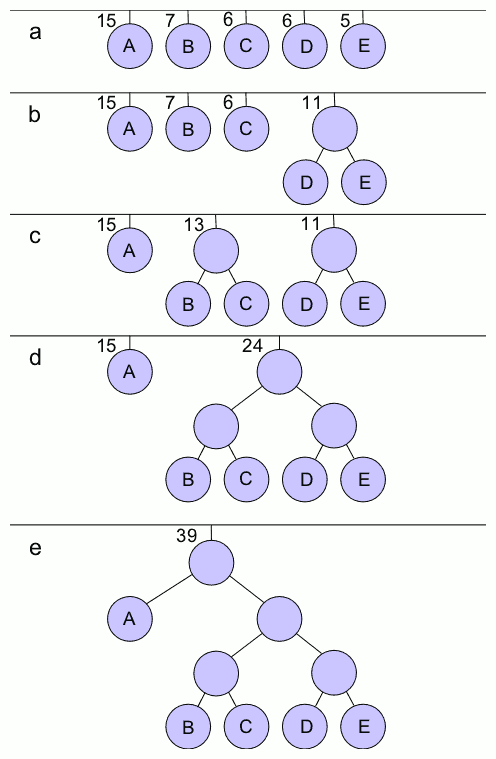
b）将D和E合并，形成了一棵权值为11的树。

c）将权值最小的两个B和C (分别为7和6)合并，合并为一棵权值为13的树。

d）合并的含有BC和含有DE的两个树，形成了一棵权值为24的树。

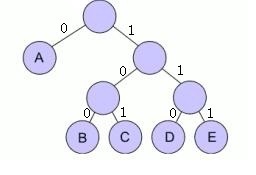
e)将仅剩的两棵树合并，形成了一棵权值为39的树

图2.9 哈夫曼树的构建过程



然后可以按上图安排每条树边的编码 A-0,B-100,C-101,D-110,E-111，如图2.11，这样平均编码长度就是(1+3+3+3+3)/5=13/5=2.6

图2.10 哈夫曼编码



如何每次取出最小的两个数，再加入一个数，可以采用优先队列（后面学习）实现。另外这里介绍一个更加简单的方法：维护两个有序队列。具体做法是，初始化两个队列queue1和queue2，其中queue1为初始的从小到大排序的n个数，queue2为空。然后从queue1和queue2的队首中取出2个最小的数，相加后添加到queue2的队尾，重复上述操作n-1次，直到两个队列只有1个数为止。（为什么能这么做，请读者思考）

建议需要建哈弗曼编码的实例（哈弗曼树的构建代码可以借用前面的，不要重复）

**§2.6二叉堆及其应用**

**一、什么是二叉堆**

二叉堆是具有堆性质的一棵二叉树。对于每一个节点，它的权值是以该节点为根的子树最小值（小堆）或最大值（大堆）。如图2.11

由定义可以知堆的最小值或最大值一定在根节点上。

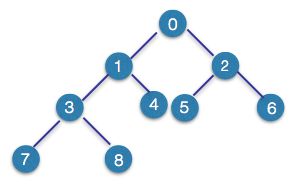


图2.11 二叉堆示例

**二、 二叉堆构造及其基本操作**

二叉堆有两个重要的操作：插入一个元素和删除堆顶元素。下面以小根堆为例讲述其操作实现。

因为二叉堆是一棵完全二叉树，因此节点x的父亲是x/2,节点x的儿子是2x和2x+1。

**插入一个元素进堆**：

一棵有n-1个节点的堆，现在要插入第n个节点。先把插入的元素放在数组的第n个位置上。这时完全二叉树不满足堆性质，此时需要将它调整为堆。

从下往上调整。不断将当前节点和父亲比较，如果该节点较小，就将它与父节点交换，重复这个操作直到不能调整为止。

void insert(int n, int val) { // 插入第n个元素，它的值是val

a[n] = val;

while (n / 2 && a[n / 2] > a[n]) {

swap(a[n / 2], a[n]);

n /= 2;

}

}

**删除堆顶元素：**

删除根节点，这时第1个位置就成了空位。首先将最后一个节点n移动到第1个节点，此时该完全二叉树不满足堆性质，此时需要将它调整为堆。

从上往下调整。从根节点开始，不断将当前节点与左右孩子比较，若该节点较大，则将左右孩子较小的节点与之交换，重复这个操作直到不能调整为止。

void delete(int n) { // 现在堆里还有n个元素

a[1] = a[n]; n--; //删除1个元素，堆中只有n-1个元素了

int x = 1;

while (2 \* x <= n) {

int y = 2 \* x;

if (y + 1 <= n && a[y] > a[y + 1]) y++;

if (a[x] > a[y]) {

swap(a[x], a[y]);

x = y;

} else break;

}

}

由于堆是完全二叉树，无论向上还是向下调整操作，最多调整log2 n次，因此时间复杂度为O(log2 n)。

**三、二叉堆的STL实现**

因为堆的作用主要用来获取最小/大值，类似队列的取最值操作，因此堆有一个别名叫做优先队列。在STL中提供了“优先队列”模板，英文priority\_queue。

使用STL需要在代码开头添加如下两行：

#include<queue>

using std::priority\_queue;

第一行是包含了优先队列的头文件，第二行是调用std的名字空间。

例如，声明一个存储int类型的优先队列Q，默认为大堆。代码如下：

priority\_queue<int> Q;

下面是代码说明优先队列的一些常用函数：

inline void test()//表示这是一个用来测试的函数

{

int x=3;

Q.push(x);//将3插入到堆中

Q.push(5);//将5插入堆中

Q.push(x\*2);//将6插入堆中

x=Q.top();//取堆顶元素，这里应该是6

Q.pop(); //删除堆顶元素，这里是删除6

x=Q.size();//求堆中元素的总个数，这里应该为2

while(!Q.empty()) Q.pop(); //若优先队列还存在元素，则反复删除堆顶，实际上起到了清空堆的作用。这里就是依次把5和3删除

return;  
}

值得注意的是，stl的优先队列有一点和日常观念不一样：它只能支持访问最大的元素，也就是说，它是一个大根堆。如果你只是想用小根堆，那么你不妨将所有数乘以-1以后再插入，这样就可以得到想要的结果了。

**四、二叉堆的应用**

**例2.5** 初始一个小根堆为空，需要支持以下3种操作：

操作1： 1 x 表示将x插入到堆中

操作2： 2 输出该小根堆内的最小数

操作3： 3 删除该小根堆内的最小数

输入格式：

第一行包含一个整数N，表示操作的个数

接下来N行，每行包含1个或2个正整数，表示三种操作，格式如下：

操作1： 1 x

操作2： 2

操作3： 3

输出格式：

包含若干行正整数，每行依次对应一个操作2的结果。

**解答：**这道题考察的都是堆的基本操作。这里使用STL的优先队列,代码如下：

#include<cstdio>

#include<queue>

using std::priority\_queue;

priority\_queue<int> Q;

int n;

signed main()

{

int opt,x;

scanf("%d",&n);

while(n--)

{

scanf("%d",&opt);

if(opt==1)

scanf("%d",&x),Q.push(-x);//注意因为要是小根堆，所以乘以-1再插入

else if(opt==2)

printf("%d\n",-Q.top());//同样乘以-1再输出才能得到真实值

else

Q.pop();

}

return 0;

}

**例2.6** 合并果子[[1]](#footnote-0)

【题目描述】

在一个果园里，多多已经将所有的果子打了下来，而且按果子的不同种类分成了不同的堆。多多决定把所有的果子合成一堆。

每一次合并，多多可以把两堆果子合并到一起，消耗的体力等于两堆果子的重量之和。可以看出，所有的果子经过n-1次合并之后，就只剩下一堆了。多多在合并果子时总共消耗的体力等于每次合并所耗体力之和。

因为还要花大力气把这些果子搬回家，所以多多在合并果子时要尽可能地节省体力。假定每个果子重量都为1，并且已知果子的种类数和每种果子的数目，你的任务是设计出合并的次序方案，使多多耗费的体力最少，并输出这个最小的体力耗费值。

例如有3种果子，数目依次为1，2，9。可以先将1、2堆合并，新堆数目为3，耗费体力为3。接着，将新堆与原先的第三堆合并，又得到新的堆，数目为12，耗费体力为12。所以多多总共耗费体力=3+12=15。可以证明15为最小的体力耗费值。

【输入格式】

输入包括两行，第一行是一个整数n(1<＝n<=10000)，表示果子的种类数。第二行包含n个整数，用空格分隔，第i个整数ai(1<＝ai<=20000)是第i种果子的数目。

【输出格式】

输出包括一行，这一行只包含一个整数，也就是最小的体力耗费值。输入数据保证这个值小于231。

【样例输入】

3

1 2 9

【样例输出】

15

**解法：**原本的一堆石子，经过了几次合并，它的大小就对答案进行了几次贡献。要最小化每堆原本的石子的贡献之和，叶节点对应的权值，就是一堆石子的大小。其到根的路径长度，就是这堆石子到最合并次数。本题相当于求带权路径长度之和最小，即哈夫曼树的构建。

对于如何选出两个最小值和添加当前值，这里采用的是优先队列的方法（详见下一节）。

**实现代码**

#include <bits/stdc++.h>

#define LL long long

using namespace std;

const int N = 10000 + 5;

int n, cnt;

int a[N];

void INS(int n, int val) { // 插入第n个元素，它的值是val

a[n] = val;

while (n / 2 && a[n / 2] > a[n]) {

swap(a[n / 2], a[n]);

n /= 2;

}

}

void DEL(int n) { // 现在堆里还有n个元素

a[1] = a[n]; n--;

int x = 1;

while (2 \* x <= n) {

int y = 2 \* x;

if (y + 1 <= n && a[y] > a[y + 1]) y++;

if (a[x] > a[y]) {

swap(a[x], a[y]);

x = y;

} else break;

}

}

int main() {

scanf("%d", &n);

for (int i = 1; i <= n; i++) scanf("%d", &a[i]);

for (int i = 1; i <= n; i++) INS(i, a[i]);

LL ans = 0; cnt = n;

for (int i = 1; i < n; i++) {

int tmp = 0;

tmp += a[1]; DEL(cnt); cnt--;

tmp += a[1]; DEL(cnt); cnt--;

INS(++cnt, tmp);

ans += tmp;

}

printf("%lld\n", ans);

return 0;

}

**例2.7** 超级钢琴[[2]](#footnote-1)

**题目描述**

小Z是一个小有名气的钢琴家，最近C博士送给了小Z一架超级钢琴，小Z希望能够用这架钢琴创作出世界上最美妙的音乐。

这架超级钢琴可以弹奏出n个音符，编号为1至n。第i个音符的美妙度为Ai，其中Ai可正可负。

一个“超级和弦”由若干个编号连续的音符组成，包含的音符个数不少于L且不多于R。定义超级和弦的美妙度为其包含的所有音符的美妙度之和。两个超级和弦被认为是相同的，当且仅当这两个超级和弦所包含的音符集合是相同的。

小Z决定创作一首由k个超级和弦组成的乐曲，为了使得乐曲更加动听，小Z要求该乐曲由k个不同的超级和弦组成。定义一首乐曲的美妙度为其所包含的所有超级和弦的美妙度之和。小Z想知道他能够创作出来的乐曲美妙度最大值是多少。

**输入格式:**

输入第一行包含四个正整数n, k, L, R。其中n为音符的个数，k为乐曲所包含的超级和弦个数，L和R分别是超级和弦所包含音符个数的下限和上限。

接下来n行，每行包含一个整数Ai，表示按编号从小到大每个音符的美妙度。  
 **输出格式:**

输出只有一个整数，表示乐曲美妙度的最大值。  
 **样例输入**

4 3 2 3

3

2

-6

8

**样例输出**  11

**数据范围**

| 测试点 | N | k |
| --- | --- | --- |
| 1 | ≤10 | ≤100 |
| 2 | ≤1,000 | ≤500,000 |
| 3 | ≤100,000 | =1 |
| 4 | ≤10,000 | ≤10,000 |
| 5 | ≤500,000 | ≤10,000 |
| 6 | ≤80,000 | ≤80,000 |
| 7 | ≤100,000 | ≤100,000 |
| 8 | ≤100,000 | ≤500,000 |
| 9 | ≤500,000 | ≤500,000 |
| 10 | ≤500,000 | ≤500,000 |

**解答：**

把一些决策存到堆里面,然后每次选最好的,并且把差一些的放进去，实质上是从一些有序表中提取出最大的k个元素。

记一个决策为(i, l, r)意思是选择的音符序列右端点为i，左端点可以在[l, r]之间，最优值为v。

开始枚举所有的i按照题目中给的L，R把初始的不超过n的决策都扔进堆里，然后每次取出最大的，答案上累加v，然后把[l, r]裂解成[l, m-1], [l, m+1]，这里m是使之取得最优值的区间左端点。然后取k个出来就行了。

**例2.9**种树**[[3]](#footnote-2)**

**题目描述**

A城市有一个巨大的圆形广场，为了绿化环境和净化空气，市政府决定沿圆形广场外圈种一圈树。园林部门得到指令后，初步规划出n个种树的位置，顺时针编号1到n。并且每个位置都有一个美观度Ai，如果在这里种树就可以得到这Ai的美观度。但由于A城市土壤肥力欠佳，两棵树决不能种在相邻的位置（i号位置和i+1号位置叫相邻位置。值得注意的是1号和n号也算相邻位置！）。最终市政府给园林部门提供了m棵树苗并要求全部种上，请你帮忙设计种树方案使得美观度总和最大。如果无法将m棵树苗全部种上，给出无解信息。

**输入格式**

输入的第一行包含两个正整数n、m。第二行n个整数Ai。  
 **输出格式**

输出一个整数，表示最佳植树方案可以得到的美观度。如果无解输出“Error!”，不包含引号。  
 **样例输入1**

7 3

1 2 3 4 5 6 7

**样例输出1**  
 15

**样例输入2**

7 4

1 2 3 4 5 6 7

**样例输出2** Error!

**数据范围**

| 测试点编号 | n | 测试点编号 | n |
| --- | --- | --- | --- |
| 1 | 30 | 11 | 200 |
| 2 | 35 | 12 | 2007 |
| 3 | 40 | 13 | 2008 |
| 4 | 45 | 14 | 2009 |
| 5 | 50 | 15 | 2010 |
| 6 | 55 | 16 | 2011 |
| 7 | 60 | 17 | 2012 |
| 8 | 65 | 18 | 199999 |
| 9 | 200 | 19 | 199999 |
| 10 | 200 | 20 | 200000 |

对于全部数据：m≤n，-1000≤Ai≤1000

**解答：**

首先考虑如果没有“相邻位置不能都种”这一限制会怎么样。这时就是一个裸的贪心——按照A[i]从大到小排序，然后取前M个。

那么加上限制以后会发生什么呢？

假设A[3]最大，那就试图去选A[3]。选中之后首先要去掉3，并且，A[2]和A[4]也都不能选了，所以将它们删掉。

但是慢着！这可能会导致问题。假设A[3]=20,A[2]=A[4]=19，那么同时选A[2],A[4]可能比选A[3]要优！在最后的方案中可能是A[2]+A[4]而非A[3]。这种情况要怎么解决呢？

可以发现一点：由于A[3]最大，所以在最后的方案中，不可能只选A[2],A[4]中的一个。

原因很简单：假设在最优方案中选了A[2]但未选A[4]，那可以简单地把A[2]换成A[3]，由于未选A[4]，所以这样不会产生任何矛盾，并且把A[2]换成A[3]后，总的美观度不会下降。

因此，先去掉2,3,4，然后加入一个新的“物品”，其权值为A[2]+A[4]-A[3]，代表同时选2,4，删去3。这样，在选了3之后再选这个新物品，功效就相当于刚才所说的，把A[3]换成A[2]+A[4]。

这个新物品应该放在哪里呢？它的含义是“选2,4”，所以很容易想到，应该把它放在1,5中间。

出于方便起见，不妨在删掉2,4后直接把A[3]改成A[2]+A[4]-A[3]，显然这个位置是正确的。

如此就将N个物品，需要选M个的问题转化成了在N-1个物品中选的问题。并且可以发现一个很好的性质：新的3所对应的仍然是“选中物品数+1”！（把选3换成了选2,4，即多选了一个物品）

也就是说，完全可以把新的3看做一个和1，5毫无区别的物品，现只需要在1,3,5三个物品中选择M-1个！如此下去，直到选择M次，就可以得到答案。

**算法如下：**

以A[i]为关键字建大根堆，用一个链表存放当前物品。

最初链表中元素是1~N，i的后继是i+1，前驱是i-1（当然，1的前驱是N，N的后继是1）。

执行M次操作，每一次操作都将堆顶元素k取出，ans+=A[k]。然后在链表中删除k的前驱pre和后继nxt，令A[k]=A[pre]+A[nxt]-A[k]，并更新堆。

这个算法运行的很好，但你可能感觉有点虚——为什么每次选A值最大的就正确呢？

可以发现，在上面的讨论中“选3”时，实际上做的是声明如下事实：

在最终答案中要么选了3，要么同时选了2,4.换句话说，要么选了3，要么在此基础上选了A[2]+A[4]-A[3]。

所以实际上是重写了这个问题，将其变成“N-2个物品中选M-1”个的形式，如此一直化归，直到最后变成“N-2(M-1)个物品中选1个”，这时答案就是显然的。

**§2.7二叉排序树及其应用**

1. **二叉排序树的概念**

二叉排序树（Binary Sort Tree），又称[二叉查找树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%9F%A5%E6%89%BE%E6%A0%91/7077965" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)（Binary Search Tree），亦称[二叉搜索树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%90%9C%E7%B4%A2%E6%A0%91/7077855" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/_blank)。是指一棵空树或者具有下列性质的二叉树：

任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；

任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；

任意节点的左、右子树也分别为二叉查找树；

没有键值相等的节点。如图2.12

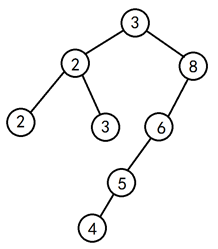


图2.12 二叉排序树示例

二叉排序树相比于其他数据结构的优势在于查找、插入的时间复杂度为O(log2 n)。二叉排序树是基础性数据结构，用于构建更为抽象的数据结构，如集合、multiset、关联数组等。

二叉排序树的查找过程和次优二叉树类似，通常采取二叉链表作为二叉排序树的存储结构。中序遍历二叉排序树可得到一个关键字的有序序列，一个无序序列可以通过构造一棵二叉排序树变成一个有序序列，构造树的过程即为对无序序列进行查找的过程。每次插入的新的节点都是二叉排序树上新的叶子节点，在进行插入操作时，不必移动其它节点，只需改动某个节点的指针，由空变为非空即可。搜索、插入、删除的复杂度等于树高，期望O(log2 n)，最坏O(n)（数列有序，树退化成线性表）。

虽然二叉排序树的最坏效率是O(n),但它支持动态查询，且有很多改进版的二叉排序树可以使树高为O(log2 n),如Treap、Splay、SBT、AVL树、红黑树等。故不失为一种好的动态查找方法。

**二、二叉排序树的查找**

在二叉排序树b中查找x的的算法：

1.若b是空树，则搜索失败，否则：

2.若x等于b的根节点的数据域之值，则查找成功；否则：

3.若x小于b的根节点的数据域之值，则搜索左子树；否则：

4.查找右子树。

**三、二叉排序树的插入**

向二叉排序树b中插入一个节点s的算法：

1.若b是空树，则将s所指节点作为根节点插入，否则：

2.若s->data等于b的根节点的数据域之值，则返回，否则：

3.若s->data小于b的根节点的数据域之值，则把s所指节点插入到左子树中，否则：

4.把s所指节点插入到右子树中。（新插入节点总是叶子节点）

**四****、二叉排序树的删除**

在二叉查找树删去一个节点\*p，分三种情况讨论：

1.若\*p节点为叶子节点，即PL（左子树）和PR（右子树）均为空树。由于删去叶子节点不破坏整棵树的结构，则只需修改其双亲节点的指针即可。

2.若\*p节点只有左子树PL或右子树PR，此时只要令PL或PR直接成为其双亲节点\*f的左子树（当\*p是左子树）或右子树（当\*p是右子树）即可，做此修改也不破坏二叉查找树的特性。

3.若\*p节点的左子树和右子树均不空。在删去\*p之后，为保持其它元素之间的相对位置不变，可按中序遍历保持有序进行调整，可以令\*p的直接前驱（in-order successor）替代\*p，然后再从二叉查找树中删去它的直接前驱。直接前驱就是指：从\*p的左儿子开始，如果有右儿子，则不断向右继续查找，否则就找到了直接前驱。由于它的直接前驱最多只有一个儿子，因此可以按照方法2删除。如图2.13

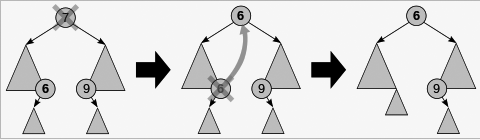


图2.13 二叉排序树删除示例

**四、二叉排序树的简单平衡方法**

由于二叉排序树的复杂度很容易退化，因此在实际中的用途没有那么广。但是如果在二叉排序树能实现平衡，那么二叉排序树的时间复杂度为O(log2 n)，非常优秀，可以得到非常广泛的应用。目前最主要的平衡方法有“AVL”、“Treap”、“Splay”等，该方法比较通用，有兴趣的同学可以自行查找资料学习。这里主要介绍另一种简单而且高效的平衡方法——“替罪羊树”。

替罪羊树的一个显著特点是：对于每一个节点，它的左儿子和右儿子的子树大小都不超过本身子树大小的0.75倍。那么树的高度是接近log2 n。但是如何做到这一点呢？

对于插入的过程，像普通二叉排序树一样插入，完成后从新节点开始向父亲查找，找到深度最小的“不平衡”的节点，“不平衡”的意思是左儿子或右儿子的子树大小超过本身子树大小的0.75倍。当然，如果没有这样的点，直接结束即可。否则，记录这个点为x。

对于删除的过程，可以发现必然会删除一个只有一个儿子的点，删除完成后从那个点的儿子开始向父亲查找，做与插入操作同样的事情，找到节点x。

插入/删除操作完成后，如果x是存在的，那么中序遍历整棵x的子树，得到这棵子树对应的序列。再O(n)进行一次暴力重构，将x的子树改造成最平衡的样子（只需要每次取最中间的数字当做根即可）。用这棵子树取代原来x的子树的位置。

虽然这样的时间复杂度看上去非常高，但实际上，用“均摊分析”等较高级的方法进行分析，可以得到一个结论：无论是怎样的输入数据，替罪羊树都可以在O(n\*log2 n)的时间内完成给定的n次操作（证明略），这样做效率确实较高。

建议增加二叉排序树的实例

1. 该题来源于NOIP2004提高组 [↑](#footnote-ref-0)
2. 该题来源于NOI2010 [↑](#footnote-ref-1)
3. 该题来源于2011国家集训队 [↑](#footnote-ref-2)