树、二叉树与堆

从线性排队走向树形分支、递归遍历与二叉堆贪心

HGNU-ACM 实验室算法组

配套：树的存储与概念 + 遍历还原 + 哈夫曼贪心 + 二叉堆上浮下沉 + 3 道真题精讲

本章目录

树与二叉树基本概念与存储

二叉树先中后序遍历与还原

哈夫曼贪心合并代价

第 1–3 节：从第 2 章队列承接树形分支存储、先中后序递归还原，以及哈夫曼贪心合并思想。

本章目录（续）

二叉堆（Heap）原理与上浮下沉

三大树、二叉树与堆真题精讲

常见错误、检查清单与练习安排

第 4-6 节：彻底掌握完全二叉树堆的 $O(\log n)$ 上浮下沉、3 道真题逐行手推与收尾清单。

承接第 2 章：当数据不再排成“一字长蛇”

- 第 1、2 章的栈和队列都是线性结构：每个元素前面最多 1 个前驱，后面最多 1 个后继。
- 现实里很多关系会分叉：电脑文件夹嵌套、家族族谱、淘汰赛对阵表，一个上级下会有多个下级。
- 先看一个手算小例子：一个 5 人的小家谱——老祖宗 a 生了 b，c，b 又生了 d，e。

写程序前先问三个问题

1. 谁是最顶端的起点？怎样区分谁是谁的孩子？
2. 如果一个点有好几个孩子，普通数组怎么装得下？
3. 从某个人出发，怎么走到他的爸爸、儿子甚至堂兄弟？

本页要点

树把“只能往前后走”的一维排队长龙，升级成“一层层向下分叉”的层次结构！

为什么只用一维列表查祖宗/子树会 TLE？

假设公司有 $n = 10^5$ 名员工，回答 $q = 10^5$ 次“x 属于哪位总监手下”

- 暴力做法：只用一张表格每行记（`员工，直接领导`），每次提问都从头扫描表格找领导，直到找到最高层。
- 操作次数估算：扫一次表格找领导最多 $O(n)$ 步，若链长 n 层再乘提问次数，总操作达 $10^5 \times 10^5 = 10^{10}$ 次，远超 1 秒限制（约 10^8 次），必然 **TLE**！
- 树形存储：让每个节点直接记住自己的孩子下标（邻接表），找下属无需全表乱扫，沿着边走 $O(1)$ 直达！

本页要点

把“谁连着谁”存成树的边表后，找孩子和子树统计不用全局扫，只走实际存在的连线！

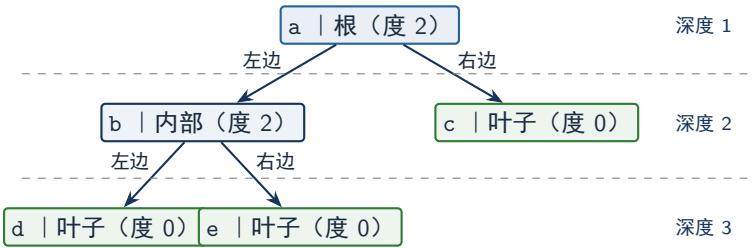
新术语大白话：根、叶子、度、深度与二叉树

- 树（**Tree**）：n 个点用 $n-1$ 条边连成一片且没有圈（无环）的图。
- 根（**Root**）：整棵树最顶端的老祖宗（没有父节点）；叶子（**Leaf**）：没有孩子的末端节点。
- 度（**Degree**）：一个节点直接往下分出几个孩子，就说它的度是多少（叶子的度为 0）。
- 深度（**Depth**）：从根走到它经过的层数（习惯设根的深度为 1，孩子深度 $=$ 父亲深度 $+1$）。
- 二叉树（**Binary Tree**）：每个节点最多只有 2 个孩子，并且严格区分左孩子和右孩子（即使只有一个孩子也要分清左或右）。

本页要点

普通树不区分孩子左右顺序；二叉树哪怕只有一个孩子，放左边和放右边也是两棵不同的树！

图解：5 点树上的根、父子、叶子、度与深度



本页要点

树共有 5 个点、4 条边；所有节点深度最大值 3 就是这棵树的高度（深度）。

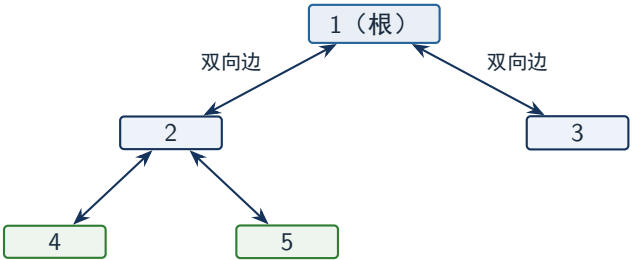
两种存树法：邻接表存普通树 vs 数组编号存二叉树

存储方式	写法与内存特征	适用场景与注意
邻接表 g[u]	vector<int> g[N] 存每个点连出去的邻居	适合孩子数量不定的普通树；若要向上走必须存双向边
左右儿子数组	lch[u]，rch[u] 存左/右孩子编号，0 表空	适合一般形态二叉树（可能只有左链或右链）
完全二叉树编号	根为 1，点 i 左孩子 2*i、右孩子 2*i+1	只适合完全二叉树（如二叉堆），偏斜链会浪费 2^n 空间！

本页要点

普通树和要“回头找父亲”的题目用邻接表存双向边；堆这种紧凑二叉树用 2i，2i+1 一维数组最省事！

图解（1/2）：邻接表存双向树（可往返）

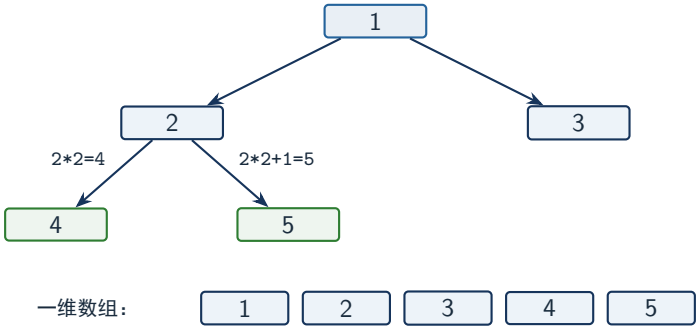


邻接表：`g[1]={2,3}`，`g[2]={1,4,5}`，`g[3]={1}`，`g[4]={2}`，`g[5]={2}`

本页要点

普通树或需要“从孩子往回找父亲”的题目，用邻接表存双向边，遍历时传 `fa` 防走回头路即可。

图解（2/2）：完全二叉树 2i，2i+1 一维数组编号



本页要点

为什么不能用 2i，2i+1 存任意二叉树？若 100 个点全只有右孩子，第 100 个点编号会达到 $2^{100}-1$，立刻内存爆炸！

手算跟踪表：用第 2 章队列给 5 点树做层序遍历（BFS）

对左图 5 点树（1-{2,3}, 2-{1,4,5}），从根 1 出发用队列求每个点到根的距离 `dist[]`：

轮次	弹出队头 u	新入队的未访问邻居	操作后队列 q	更新的 dist
初始化	—	放入起点 1	[1]	dist[1]=0
第 1 轮	1	2，3	[2，3]	dist[2]=1，dist[3]=1
第 2 轮	2	4，5（1 已访问跳过）	[3，4，5]	dist[4]=2，dist[5]=2
第 3 轮	3	无（1 已访问）	[4，5]	不变
第 4 轮	4	无（2 已访问）	[5]	不变
第 5 轮	5	无（2 已访问）	[]	不变
终止步	队列为空	!q.empty() 为假，退出	[]	全部距根距离算完

本页要点

双向邻接表一定要用 `dist[v] == -1`（或记录父亲）防止儿子走回爸爸造成死循环！

为什么“n 点 $n-1$ 边无环”能保证两点路径唯一？

- 先反过来想：如果节点 u 和 v 之间存在两条不一样的路径，把这两条路径拼在一起，立刻就能围成一个环！
- 边数的约束：把 n 个孤立点连成 1 棵树，每次加一条边把两个不连通的块合并成 1 块，连通块数量减 1；从 n 块变成 1 块恰好需要 $n-1$ 条边。
- 结论：只要连通且恰好 $n-1$ 条边，就绝对不可能有环，也就意味着任意两点之间有且只有一条简单路径（这正是第 3 章《医院设置》敢直接用 BFS 求唯一距离的依据！）。

本页要点

树上任意两点间的距离是唯一确定的，不需要像一般图那样比较“哪条路线更短”！

模板代码：邻接表双向存树与 BFS 求距离

```
vector<vector<int>> g(n + 1);                          // 1-based 邻接表存双向边
for (int u = 1, l, r; u <= n; ++u) {
    cin >> l >> r;                                     // 读左右儿子，0 表示空
    if (l != 0) { g[u].push_back(l); g[l].push_back(u); }
    if (r != 0) { g[u].push_back(r); g[r].push_back(u); }
}
vector<int> dist(n + 1, -1); queue<int> q;             // -1 表示尚未访问过
dist[1] = 0; q.push(1);
while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int v : g[u]) if (dist[v] == -1) {
        dist[v] = dist[u] + 1; q.push(v);              // 儿子深度 = 父亲 + 1
    }
}
```

本页要点

遇到 0 必须跳过不连边，否则所有叶子都会连向虚拟的 0 号点！

上机验证与反事实追问：建树存边与单点边界

3 组本地测试数据与预期 dist[1..n]

- 普通 **5** 点树：n=5，点 1 连 2,3，点 2 连 4,5，其余 0 0 ⇒ 输出 0 1 1 2 2。
- 偏斜链：n=3，点 1 连 2 0，点 2 连 3 0，点 3 连 0 0 ⇒ 输出 0 1 2。
- 最小单点树：n=1，点 1 连 0 0 ⇒ 队列弹出 1 后为空，正确输出 0。

反事实追问

如果只写 g[u].push_back(l) 没写反向边 g[l].push_back(u)，当从 2 号点出发 BFS 时，能到达 1 和 3 吗？不能！dist[1] 和 dist[3] 会一直保持 -1！

本页要点

如果题目需要从任意节点出发走遍全树（而不只是从根往下），必须建无向双向边！

树与存储高频易错对照（WA / RE / TLE）

错误表现与后果	错误写法	正确写法与原因
把 0 号点连进树（**WA**）	直接 `g[u].push_back(l)` 没判 `l != 0`	所有叶子全连到 0 号点形成大环，距离全错
只建单向边（**WA**）	仅连 `u -> l`，`r`，却从非根节点出发遍历	儿子走不到父亲和兄弟，只能向下不能向上
偏斜树硬用 `2i`（**RE/MLE**）	任意二叉树也开 `4*n` 堆数组按 `2i` 存	连续 30 个右儿子下标达到 2^{30}，直接越界 RE
双向边不标记访问（**TLE**）	无向图 BFS 没写 `dist[v] == -1` 判定	父子反复互入队列，死循环到超时 TLE

本页要点

记住三条底线：空儿子 0 不连边、需往返必建双向边、双向边必判 `dist[v] == -1`！

生活引入：导游带队参观“左右双院”的三种记账法

假设每个景点大门里都有一个主殿（根），左边是左偏院（左子树），右边是右偏院（右子树）。导游有三种打卡顺序：

- 先序遍历（根 $\rightarrow$ 左 $\rightarrow$ 右）：一进门先记主殿，再逛左院，最后逛右院。
- 中序遍历（左 $\rightarrow$ 根 $\rightarrow$ 右）：先逛完左院，回来拍主殿，再去逛右院。
- 后序遍历（左 $\rightarrow$ 右 $\rightarrow$ 根）：左院右院全逛完，临出门最后记主殿。

怎么快速看懂遍历序列？

只要看“根”排在第几个：排第一叫先序，夹在中间叫中序，垫底最后叫后序！

本页要点

无论是整棵大树还是里面任意一棵小子树，都严格遵守这三种“根、左、右”的固定次序！

暴力枚举所有树形为什么会 TLE？

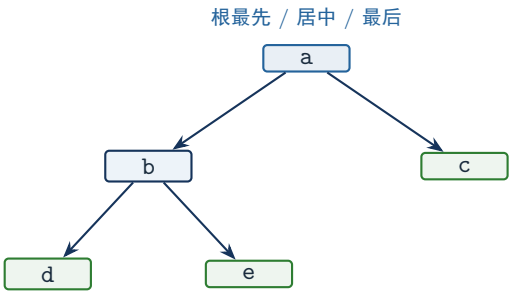
已知先序 pre 和中序 ino（长度 $n \le 26$），直接试出原树行不行？

- 暴力想法：枚举所有含 n 个点的二叉树形态，逐一算出它的先序和中序，看哪棵刚好吻合。
- 操作次数估算：n 个节点能搭出的不同二叉树数量叫卡特兰数 $C_n \approx \frac{4^n}{n^{3/2}\sqrt{\pi}}$。当 $n = 26$ 时，$C_{26} \approx 1.3 \times 10^{13}$ 种！绝对 **TLE**！
- 利用结构性质：先序第 1 个字符直接指名道姓说出“谁是根”，中序里找到这个根就能一刀劈出左子树有几个点、右子树有几个点，全程只需 $O(n)$ 递归一次！

本页要点

不靠猜，靠“先序定根 + 中序定左右规模”，每次递归直接把问题劈成规模更小的两半！

图解：同一棵 5 点树的三种遍历结果



先序（根左右）：a b d e c

中序（左根右）：d b e a c

后序（左右根）：d e b c a

本页要点

盯着根 a 看：先序里 a 在最前；中序里 a 左边是 dbe（左子树），右边是 c（右子树）；后序里 a 在最后！

三种遍历核心用途与区间新术语

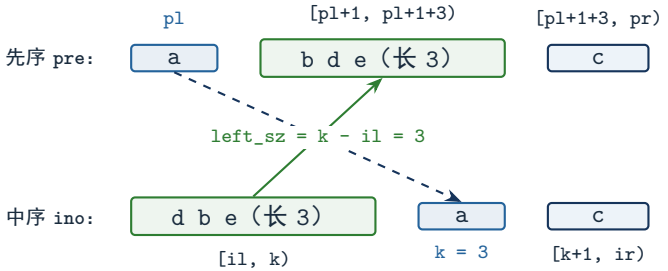
遍历名称	访问顺序	在算法题中的核心作用
先序	根 → 左 → 右	第 1 个字符永远是当前子树根节点，负责“发号施令”找根
中序	左 → 根 → 右	找到根的位置后，左侧全部属于左子树、右侧全部属于右子树
后序	左 → 右 → 根	先算完左右两半子问题，最后汇总给根（自底向上统计/还原）

- 左闭右开区间 `[pl, pr)`：包含左端 `pl`、不包含右端 `pr`，区间长度就是 `pr - pl`；当 `pl >= pr` 时表示空子树，直接 `return`！
- 左子树规模 `left_sz`：若根在中序下标为 `k`，中序起点为 `il`，则左边共有 `left_sz = k - il` 个节点。

本页要点

用左闭右开 `[l, r)` 的最大好处：区间长度直接等于 `r - l`，空区间统一是 `l >= r`！

核心图解：先序定根 + 中序劈两半的长度对齐



本页要点

先序左子树从 pl+1 开始占 left_sz 个位置，因此先序右子树必定从 pl + 1 + left_sz 开始！

手算跟踪表：build(pl, pr, il, ir) 还原 abdec + dbeac

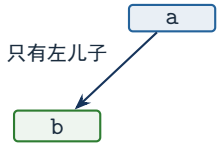
跟踪 pre="abdec"、ino="dbeac"（下标 $0 \sim 4$）递归过程及何时输出后序字符：

步骤	[pl,pr)	[il,ir)	根 root	k 与 left_sz	下一步动作 / 终止与写入 post
1	[0,5)	[0,5)	a	k=3, sz=3	拆左 [1,4),[0,3) 与右 [4,5),[4,5)
2	[1,4)	[0,3)	b	k=1, sz=1	拆左 [2,3),[0,1) 与右 [3,4),[2,3)
3	[2,3)	[0,1)	d	k=0, sz=0	左右皆空（pl>=pr 返回），写 d
4	[3,4)	[2,3)	e	k=2, sz=0	左右皆空返回，写 e；回 b 写 b
5	[4,5)	[4,5)	c	k=4, sz=0	左右皆空返回，写 c
终止	—	—	—	回到第 1 步末尾	写入最外层根 a，得 debca

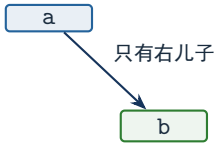
本页要点

每次先把左、右子调用彻底走完（包含空子树 pl>=pr 立即返回），最后一行才 push_back(root)！

为什么“先序 + 中序”唯一，而“先序 + 后序”不行？



先序 ab | 后序 ba | 中序 ba



先序 ab | 后序 ba | 中序 ab

- 只给先序 ab 和后序 ba，两棵树输出完全一样，根本分不清 b 在左还是在右！
- 加入中序后：左图 b 在 a 左边（ba），右图 b 在 a 右边（ab），瞬间唯一确定！

本页要点

同理：“后序 + 中序”也能唯一还原二叉树，但“先序 + 后序”除非每个非叶点都有俩孩子，否则不唯一！

模板代码：先序 + 中序递归求后序

```
string pre, ino, post;
int pos[256];                                          // pos[c] 存字符 c 在中序的下标
void build(int pl, int pr, int il, int ir) {
    if (pl >= pr) return;                              // 空区间直接返回（递归边界）
    char root = pre[pl];                               // 先序首字符必是当前子树的根
    int k = pos[(unsigned char)root];                  // 在中序中定位根，O(1) 查表
    int left_sz = k - il;                              // 左子树包含的节点个数
    build(pl + 1, pl + 1 + left_sz, il, k);            // 1. 递归左子树
    build(pl + 1 + left_sz, pr, k + 1, ir);            // 2. 递归右子树
    post.push_back(root);                              // 3. 左右全算完再输出根（后序）
}
```

本页要点

提前用 pos[] 预处理字符位置，每次定位根只需 $O(1)$，整棵树还原总时间复杂度 $O(n)$！

上机验证与反事实追问：三种极端子树形状

3 组本地验证数据与预期输出

- 普通树：输入 `abdec / dbeac` ⇒ 输出 `debca`。
- 全左偏斜链：输入 `abc / cba` ⇒ 右子树每次规模都为 0，输出 `cba`。
- 单节点树：输入 `a / a` ⇒ `left_sz=0`，左右均命中 `pl>=pr` 返回，输出 `a`。

反事实追问

如果把右子树起点写错成 `build(pl + left_sz, pr, k + 1, ir)`（漏了 `+ 1`），会发生什么？先序的根后面漏算了 1 格，右子树会抢走左子树最后一个字符 `e`，导致死循环或完全错误的输出！

本页要点

每次自己写完 `build`，拿全左链 `abc/cba` 和全右链 `abc/abc` 验一遍，能立刻抓出边界差 1 的错！

二叉树遍历还原高频易错对照（WA / RE）

错误表现与后果	错误写法	正确写法与原因
右段起点少加 1（**WA**）	写成 `pl + left_sz`	左段起点是 `pl+1`、长 `left_sz`，右段必从 `pl+1+left_sz` 起
漏写边界判空（**RE**）	忘记 `if (pl >= pr) return;`	空子树无限递归导致栈溢出 **RE**
混用开闭区间（**WA**）	传 `n - 1` 当右端点却写 `pl >= pr`	统一用左闭右开 `[0, n)`，单点时 `0 < 1` 才会进入
`char` 负下标（**RE**）	直接写 `pos[root]` 遇到扩展字符	转成 `pos[(unsigned char)root]` 最安全

本页要点

“左闭右开 `[0, n)`” + “`pl >= pr` 返回” + “`pl + 1 + left_sz`”，这三处对齐就永不差 1！

生活引入：搬果子体力怎么省？

院子里放着 3 堆果子，重量分别是 [1，2，9]。每次只能把两堆合到一起，花费体力等于两堆重量之和：

- 随手先合大的：先把 2 和 9 合成 11（耗力 11），再把 11 和 1 合成 12（耗力 12），总共花了 $11 + 12 = \mathbf{23}$。
- 先挑最小的合：先把 1 和 2 合成 3（耗力 3），再把 3 和 9 合成 12（耗力 12），总共只花了 $3 + 12 = \mathbf{15}$！

关键发现

重量高达 9 的那堆，在错法里被搬了 **2** 次，在正解里只被搬了 **1** 次！越重越要晚点动，越轻越要先抱团！

本页要点

总耗力 = 每堆果子的“重量 × 它参与合并的次数”之和，因此最重果子绝不能放在最深层！

暴力枚举为什么 TLE？只排一次序为什么 WA？

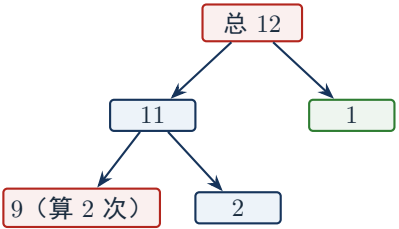
- 暴力枚举所有合并次序（**TLE**）：n 堆每次挑两堆合，不同合并方案数为 $(2n-3)!!$。当 $n = 3 \times 10^4$ 时，天文数字直接 **TLE**！
- 常见误区：只 `sort` 一次从头加到尾（**WA**）：来看反例 `[3, 4, 5, 6]`！
  - 只排一次序错法：先合 3+4=7，接着硬把新堆 7 和 5 合并（耗 12），再合 12+6=18，总耗力 $7 + 12 + 18 = \mathbf{37}$！
  - 哈夫曼正解：合完 3+4=7 放回去后，当前最小两堆是 5 和 6！先合 5+6=11，再合 7+11=18，总耗力 $7 + 11 + 18 = \mathbf{36}$！

本页要点

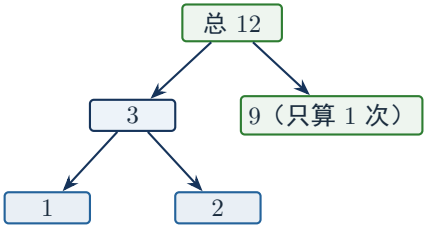
合并产生的新果子堆可能比后面的旧堆还大，必须放回集合里重新参与“挑最小两堆”！

新术语大白话 + 图解：哈夫曼树与带权路径长度

错法：重果子 9 在深层（耗 23）



哈夫曼正解：1，2 在深层（耗 15）



- 哈夫曼树：把每次合并看作“用新父亲包住两个孩子”，这样合出来的最优二叉树就是哈夫曼树。
- 带权路径长度（**WPL**）：所有叶子“重量 $\times$ 深度（根深度记为 0）”的总和，恰好等于合并总耗力！

本页要点

右图 WPL $= 1 \times 2 + 2 \times 2 + 9 \times 1 = 15$，把最轻的 1，2 放到最深处正是省体力的秘密！

手算跟踪表：用贪心规则合并 [1，2，5，9]

每轮选当前最小两堆合并，把新堆放回集合，直到只剩 1 堆：

轮次	合并前所有堆	取出最小两堆	合并新堆（本轮耗力）	累计总体力 ans
初始	{1，2，5，9}	—	—	0
第 1 轮	{1，2，5，9}	1 和 2	合成 3 放回 → {3，5，9}	$0+3=3$
第 2 轮	{3，5，9}	3 和 5	合成 8 放回 → {8，9}	$3+8=11$
第 3 轮	{8，9}	8 和 9	合成 17 放回 → {17}	$11+17=28$
终止步	{17}（仅剩 1 堆）	size > 1 为假	循环结束，不再合并	输出 28

本页要点

总共 $n=4$ 堆果子，每轮让总堆数减少 1，因此恰好循环 $n-1=3$ 轮后只剩 1 堆停止！

为什么每次合最小两堆一定最优？（交换论证）

假设最优合并树最深一层有两个兄弟叶子 a, b，其中有某片更轻的果子 x 却放在浅层（即 $w_x < w_a$，深度 $d_x < d_a$）：

- 把 x 和 a 对调位置，总耗力的变化量为：

$$\Delta = (w_x \cdot d_a + w_a \cdot d_x) - (w_x \cdot d_x + w_a \cdot d_a) = (w_a - w_x)(d_x - d_a) < 0$$

- 因为 $w_a > w_x$ 且 $d_x < d_a$，对调之后总耗力严格变小！
- 这说明任何最优树都能通过把“最轻两堆”换到最深层做兄弟而变得更好（或同样好），之后这俩兄弟合成权重为 $w_x + w_y$ 的新叶子，问题缩小成 $n-1$ 堆继续贪心！

本页要点

“重的往浅处换、轻的往深处沉”，这就是哈夫曼贪心一定全局最优的铁证！

模板代码：哈夫曼贪心合并核心循环

```
priority_queue<long long, vector<long long>, greater<long long>> pq;
for (int i = 0; i < n; ++i) { long long x; cin >> x; pq.push(x); }
long long ans = 0;
while (pq.size() > 1) {                          // 至少剩 2 堆才继续合并
    long long a = pq.top(); pq.pop();            // 弹出当前第 1 小
    long long b = pq.top(); pq.pop();            // 弹出当前第 2 小
    ans += a + b;                                // 累加本次耗费体力
    pq.push(a + b);                              // 新合成的大堆务必塞回堆中！
}
cout << ans << '\n';
```

本页要点

共 $n-1$ 轮合并，每轮 2 弹 1 推，单次堆操作 $O(\log n)$，总复杂度 $O(n \log n)$！

上机验证与反事实追问：哈夫曼边界测试

3 组本地测试数据与预期输出

- 基础 **3** 堆：`3 / 1 2 9` ⇒ 输出 15。
- 最小边界 $n = 1$：`1 / 100` ⇒ `while (pq.size() > 1)` 一次都不进入，输出 0（不用搬运，不耗体力）。
- 反例验证：`4 / 3 4 5 6` ⇒ 第 1 轮合 3+4=7 放回后取 5+6=11，再合 7+11=18，输出 36。

反事实追问

如果只把循环条件改成 `do { ... } while (pq.size() > 1);`，当输入 $n = 1$ 时会发生什么？第一次取出 a 后堆空了，第二次执行 `pq.top()` 直接访问空堆触发段错误 **RE**！

本页要点

写堆的循环永远用 `while (pq.size() > 1)` 先判够不够 2 个元素再弹出！

哈夫曼合并高频易错对照（WA / RE / 溢出）

错误表现与后果	错误写法	正确写法与原因
默写错大根堆（**WA**）	写成 `priority_queue<long long>`	STL 默认是大根堆，每次挑了最大两堆合
合完忘记放回（**WA**）	只用 `sort` 排一次顺序逐项累加	新堆重量变大后可能不再是最小的，必须放回堆重排
单堆没判空（**RE**）	`n=1` 时直接弹出两次计算	用 `while (pq.size() > 1)`，$n = 1$ 自动输出 0
答案范围溢出（**WA**）	在大权值变式题里用 `int ans`	合并体力累加极易超 2×10^9，一律开 `long long`

本页要点

小根堆三个模板参数 + `while (pq.size() > 1)` + `long long`，哈夫曼题稳过！

为什么需要自己理解二叉堆（Heap）？

刚才合并果子用了 `priority_queue`，但如果不懂它内部原理，遇到“修改堆内某个指定编号元素”或手写堆优化 Dijkstra 就会卡住！

- 暴力做法为什么 **TLE**：如果用普通数组存这 n 堆果子，每次暴力扫一遍找最小要 $O(n)$，n 次合并就是 $O(n^2) = 9 \times 10^8$ 必 **TLE**！
- 如果每次 `sort` 呢？：n 轮每次排序 $O(n \log n)$，总共 $O(n^2 \log n)$ 更慢！
- 二叉堆的威力：每次插入和弹出最小值都只走一条从根到叶的树高路径，仅需 $O(\log n)$ 步（$n = 3 \times 10^4$ 时最多走 15 步）！

本页要点

堆不是把所有数排好序，而是只保证“树根永远最小”，换取插/删只要 $O(\log n)$！

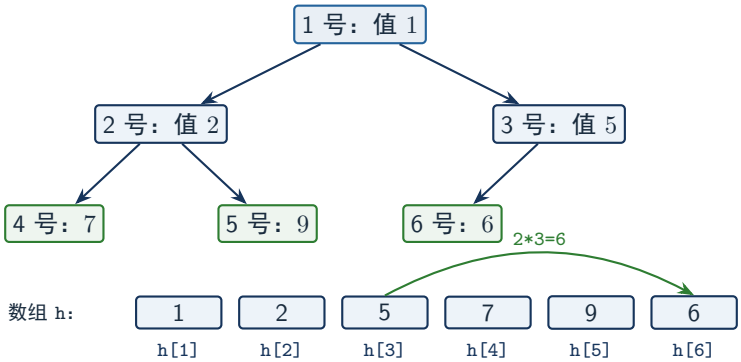
新术语大白话：完全二叉树、堆序性、小根堆/大根堆

- 完全二叉树：每一层从左到右紧凑填满、中间绝不留空洞的二叉树。正因没有空洞，才能毫无浪费地按行放进一维数组 `h[1..sz]`！
- 小根堆的“堆序性”：每一个节点的值，都小于等于它左右两个孩子的值（只要求父子有序，左右兄弟之间谁大谁小无所谓！）。
- 为什么树根 `h[1]` 必定是全堆最小？：顺着任意节点一路往上找父亲，数值越来越小，终点就是树根 `h[1]`！
- 大根堆：把“父 $\leq$ 子”反过来变成“父 $\geq$ 子”，树根 `h[1]` 就是全局最大值。

本页要点

堆的下标铁律（1-based）：点 `p` 的左儿子是 `2*p`，右儿子是 `2*p+1`，父亲是 `p/2`！

图解：完全二叉树堆与一维数组 h[1..6] 一一对应

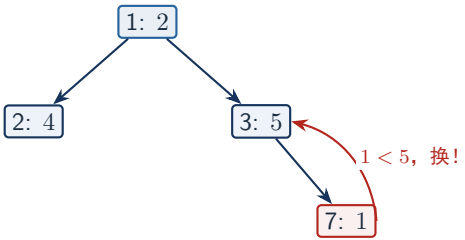


本页要点

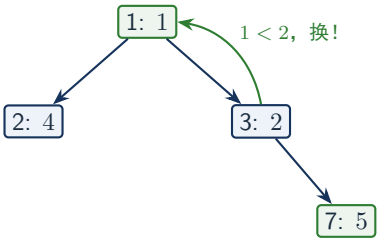
完全二叉树根本不需要开 lch/rch 指针！只需要一个一维数组 h[] 和当前大小 sz！

操作一：push(x) 插入后如何“上浮”修复堆序？

步骤 **1**：放末尾 `h[7]=1`



步骤 **2**：继续上浮换到根



- 先把新元素放到数组最后一位 `h[++sz] = x`（保持完全二叉树不留空洞！）。
- 只要当前节点 `p > 1` 且 `h[p] < h[p / 2]`（比爸爸小），就和爸爸交换，`p /= 2` 一路往上浮！

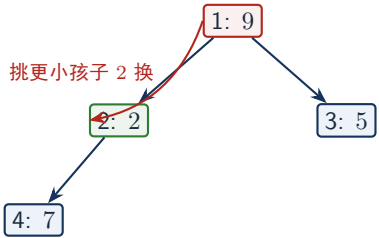
本页要点

上浮最多只走树高 $\lfloor \log_2 n \rfloor$ 步，一旦来到树根（`p == 1`）或“不再比爸爸小”立刻停下！

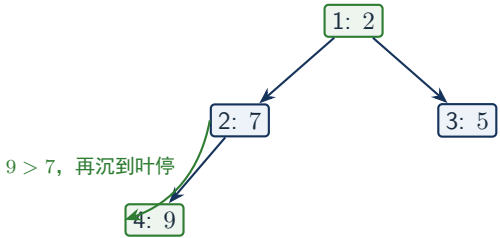
操作二：pop() 弹出最小值如何“下沉”？

为什么不能直接删掉 h[1]？因为树根会出现空洞，破坏完全二叉树编号！

1. 末尾 h[5]=9 顶替根，sz--



2. 与更小孩子交换下沉



本页要点

下沉口诀：先看左孩子 2*p 存不存在；若有右孩子且右孩子更小就选右孩子；若我比更小孩子还大就交换下沉！

手算跟踪表：依次执行 push(1) 与 pop()

初始小根堆 h = [2, 4, 5, 8, 9]（sz = 5，下标 $1 \sim 5$）：

操作与步骤	当前下标 p	对比对象与结果	动作	变化后 h[1..sz]
push(1) 第 1 步	p=6	比较父 h[3]=5：$1 < 5$	交换 h[6],h[3]	[2, 4, 1, 8, 9, 5]
push(1) 第 2 步	p=3	比较父 h[1]=2：$1 < 2$	交换 h[3],h[1]	[1, 4, 2, 8, 9, 5]
push(1) 终止步	p=1	p > 1 为假	上浮结束	[1, 4, 2, 8, 9, 5]
pop() 顶替根	p=1，sz=5	末尾 5 换到 h[1]	删去原 1	[5, 4, 2, 8, 9]
pop() 下沉 1 步	p=1	孩子 4，2 中最小是 h[3]=2	$5 > 2$ 交换	[2, 4, 5, 8, 9]
pop() 终止步	p=3	2*3=6 > sz=5，无孩子	下沉结束	[2, 4, 5, 8, 9]

本页要点

注意终止步的两种情况：上浮是 p==1 或 h[p] >= h[p/2]；下沉是无左儿子（2*p > sz）或 h[p] <= h[c]！

为什么小根堆始终满足“堆顶最小”？（不变量论证）

我们用循环不变量检查小根堆的“任意节点 u 满足 $h[u] \le h[2u], h[2u+1]$”：

阶段	要验证的内容
初始化	空堆 `sz = 0` 或只有 1 个元素时，没有违反父子关系的边，天然成立
保持（push）	上浮时若 `h[p]` 换到父节点 `p/2`，因为旧 `h[p]` $<$ 旧 `h[p/2]` $\le$ 兄弟，新父依然 $\le$ 原来两个孩子
保持（pop）	下沉时总是挑更小的孩子换上去，因此换上去的数一定 $\le$ 另一个兄弟，上面堆序不破
终止	每次操作最多走 $\log_2 n$ 步即恢复全树堆序，故 `h[1]` 永远等于当前全局最小值

本页要点

正是“下沉必须跟更小的孩子交换”，才保住了新换上去的父亲也能压住另一个兄弟！

模板代码（1/2）：手写小根堆 push 上浮

```cpp
const int MAXN = 100005;
long long h[MAXN];                          // 1-based 小根堆数组
int sz = 0;                                 // 当前堆中元素个数

void heap_push(long long x) {
    h[++sz] = x;                            // 1. 新元素先放末尾
    int p = sz;
    while (p > 1 && h[p] < h[p / 2]) {      // 2. 只要不是根且比父亲小
        swap(h[p], h[p / 2]);               //    就和父亲交换
        p /= 2;                             //    指针上移到父亲位置
    }
}
```

本页要点

数组下标从 1 开始时，父亲统一是 `p / 2`，左儿子 `2 * p`，右儿子 `2 * p + 1`，非常整齐！

模板代码（2/2）：手写小根堆 pop 下沉

```
void heap_pop() {
    h[1] = h[sz--];                          // 1. 末尾元素顶替堆顶，大小减 1
    int p = 1;
    while (p * 2 <= sz) {                    // 2. 只要至少存在左孩子
        int c = p * 2;                       //    先默认挑左孩子
        if (c + 1 <= sz && h[c + 1] < h[c])  //    若右孩子存在且更小
            c++;                             //    改挑右孩子！
        if (h[p] <= h[c]) break;             //    已经比更小孩子小，下沉结束
        swap(h[p], h[c]);
        p = c;                               //    指针下移到孩子位置
    }
}
```

本页要点

一定要先判断 `c + 1 <= sz` 才能访问 `h[c + 1]`，否则当节点只有左孩子时必越界！

STL priority_queue 与手写堆怎么选？

实现方式	写法要点	适用场景
STL 大根堆	`priority_queue<long long> pq;`	默认取最大值，代码只需 1 行
STL 小根堆	`priority_queue<long long, vector<long long>, greater<long long>> pq;`	哈夫曼合并、贪心取最小值（三参数缺一不可）
手写数组堆	用 `h[1..sz]` 配 `heap_push` / `heap_pop`	需要删堆内指定元素或理解原理时

本页要点

比赛里不改堆内元素优先用 STL `priority_queue`，但原理必须懂上浮与下沉！

上机验证与反事实追问：手写堆边界测试

3 组本地测试数据与预期输出

1. 乱序插弹：依次 push 5，2，8，1，3 后连续弹出 ⇒ 严格按 1，2，3，5，8 递增。
2. 全相等序列：依次 push 4，4，4 后弹出 ⇒ 4，4，4，验证 h[p] <= h[c] 不死循环。
3. 单点弹出：push(7) 后立即 pop() ⇒ 此时 sz=0，循环 p*2 <= 0 不进入。

反事实追问

在 heap_pop() 里，若删掉选更小孩子的 if (c + 1 <= sz && h[c + 1] < h[c]) c++;，永远只和左孩子比较，会怎样？若左孩子是 10、右孩子是 3、当前是 9，h[p] <= 10 会提前 break，导致 9 压在 3 上面破坏堆序！

本页要点

下沉时必须从左右孩子里挑出更小的那一个对比交换，才能保住另一侧兄弟！

二叉堆高频易错对照（WA / RE）

错误表现与后果	错误写法	正确写法与原因
STL 默认是大根堆（**WA**）	要最小值却写 `priority_queue<int>`	必须写全三参数带 `greater<int>` 才是小根堆
下沉漏判右儿子存在（**RE**）	不写 `c + 1 <= sz` 直接读 `h[c + 1]`	当 `sz` 为偶数时最后一个非叶点只有左儿子，会读越界
下沉未选较小儿子（**WA**）	直接拿左孩子跟父节点比较	较大数字可能压住更小的右子树，破坏堆序
空堆读堆顶或弹出（**RE**）	`sz == 0` 或 `pq.empty()` 时调 `top/pop`	访问空堆前务必先判 `!pq.empty()`（或 `sz > 0`）

本页要点

每次调 `top()` 或 `pop()` 前先看一眼堆是不是空的；手写下沉牢记“判右子、挑更小”！

例 1 | 求后序遍历（一本通 1339）[基础]：先读懂要求

大白话题意

给你一棵二叉树的先序遍历字符串和中序遍历字符串（节点都用小写字母表示），请你还原出这棵树的后序遍历字符串。

- 输入：两行字符串，第一行是先序遍历 pre，第二行是中序遍历 ino（长度不超过 26，无重复字母）。
- 处理：根据“先序定根 + 中序劈开左右子树”的递归关系，还原出后序遍历序列。
- 输出：一行字符串，表示后序遍历序列。

原题样例

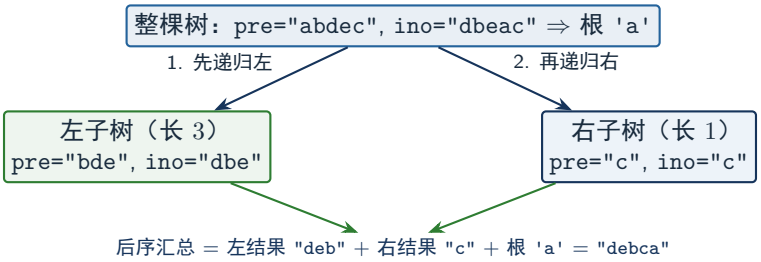
输入：abdec（换行）dbeac ⇒ 输出：debca

本页要点

节点全是互不相同的小写字母，因此字符本身就能当唯一标记，不用额外开树形结构体！

例 1 | 求后序遍历 | 怎么想到的与建模图解

暴力枚举所有树形会按卡特兰数爆炸，我们直接用第 2 节学的递归分治建模：



本页要点

只要拿到区间 [pl, pr) 和 [il, ir)，就能算出 left_sz = k - il，根本不必真的建指针！

例 1 | 求后序遍历 | 样例手推跟踪表

输入 pre="abdec", ino="dbeac"（长度 5，下标 $0 \sim 4$），初始调用 build(0, 5, 0, 5)：

进入顺序	build 参数	root	k	left_sz	左/右子调用与 post 追加
1	(0,5, 0,5)	'a'	3	3	进左 (1,4,0,3)、进右 (4,5,4,5)、最后加 'a'
2	(1,4, 0,3)	'b'	1	1	进左 (2,3,0,1)、进右 (3,4,2,3)、最后加 'b'
3	(2,3, 0,1)	'd'	0	0	左 (3,3) 空、右 (3,3) 空 $\rightarrow$ 加 'd'
4	(3,4, 2,3)	'e'	2	0	左右皆空 $\rightarrow$ 加 'e'；回第 2 步加 'b'
5	(4,5, 4,5)	'c'	4	0	左右皆空 $\rightarrow$ 加 'c'；回第 1 步加 'a'

最终 post 依次追加字符为 d -> e -> b -> c -> a，得 debca，与样例一致！

本页要点

左右子树在递归返回时已经按后序写好了，当前层只需在最后写一行 post.push_back(root)！

例 1 | 求后序遍历 | 核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
1. 预处理 `pos[ino[i]] = i`	`d:0, b:1, e:2, a:3, c:4`	用 `unsigned char` 防越界了吗？
2. 取当前根 `root = pre[pl]`	第 1 层取 `'a'`	空区间 `pl >= pr` 先返回了吗？
3. 算左子树大小 `k - il`	`3 - 0 = 3`	减的是 `il` 而不是 `0` 吗？
4. 先左再右递归，末位记录根	依次写入 `debca`	右段起点是 `pl+1+left_sz` 吗？

相对模板改了哪里？

把“建实体树再后序遍历”省成“递归末尾直接 `post.push_back(root)`”，零指针不到 20 行！

本页要点

右子树的中序起点 `il > 0`，所以 `left_sz` 务必写 `k - il`！

例 1 | 求后序遍历 | 核心代码（1/2）：递归还原函数

```
string pre, ino, post;
int pos[256];
void build(int pl, int pr, int il, int ir) {
    if (pl >= pr) return;                          // 空子树直接返回
    char root = pre[pl];                           // 先序第一项必是当前子树的根
    int k = pos[(unsigned char)root];              // 在中序中定位根
    int left_sz = k - il;
    build(pl + 1, pl + 1 + left_sz, il, k);        // 递归左子树
    build(pl + 1 + left_sz, pr, k + 1, ir);        // 递归右子树
    post.push_back(root);                          // 左右处理完再记根节点
}
```

本页要点

与 code/postorder_tree.cpp 第 $5 \sim 16$ 行一致：左闭右开区间让边界极度干净！

例 1 | 求后序遍历 | 核心代码（2/2）：主函数与全局声明

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    if (!(cin >> pre >> ino)) return 0;
    for (int i = 0; i < (int)ino.size(); ++i)
        pos[(unsigned char)ino[i]] = i;              // 预处理每个字符在中序的位置
    build(0, (int)pre.size(), 0, (int)ino.size());
    cout << post << '\n';
    return 0;
}
```

本页要点

完整可编译源码位于 `code/postorder_tree.cpp`，已通过 C++17 编译与样例实测！

例 1 | 求后序遍历 | 易错与 3 组自测边界

高频易错（对比排雷）

- 错把 `left_sz` 写成 `k`：一旦进入右子树，中序起点 `il > 0`，左子树节点数是 `k - il` 而不是 `k`！
- 先序右段起点少 `+ 1`：错写成 `pl + left_sz` 会把左子树最后一位吞进右子树导致 WA/RE。

3 组自测边界与预期输出

1. 单节点树：`a` / `a` ⇒ `a`（验证 `pl >= pr` 正常返回）。
2. 全左偏斜链：`abcd` / `dcba` ⇒ `dcba`（验证右子树全程为空时边界不越界）。
3. 全右偏斜链：`abcd` / `abcd` ⇒ `dcba`（验证左子树全程为空时 `il` 递增准确）。

本页要点

拿“全左链”和“全右链”手测一遍，任何差 1 的下标错都会当场现形！

例 2 | 合并果子（一本通 1369）[基础]：先读懂要求

大白话题意

果园有 n 堆不同种类的果子，第 i 堆数目为 a_i。每次任选两堆合并成一堆，消耗体力等于两堆数目之和。经过 $n-1$ 次合并成 1 堆，求消耗的最小总体力。

- 输入：第一行整数 n（$1 \le n \le 30000$）；第二行 n 个整数 a_i（$1 \le a_i \le 20000$）。
- 处理：每轮挑出当前数目最小的两堆合并并放回，累加每次体力（原题保证答案 $< 2^{31}$，用 `long long` 更稳妥）。
- 输出：一行一个整数，表示最小总体力。

原题样例

样例 1：`3 / 1 2 9` $\Rightarrow$ 15；　　样例 2：`10 / 3 5 1 7 6 4 2 5 4 1` $\Rightarrow$ 120。

本页要点

看清 n 达 30000：$O(n^2)$ 会超时，必须用 $O(n \log n)$ 的小根堆贪心！

例 2 | 合并果子 | 思路与小根堆建模图解

把每堆果子全部扔进一个小根堆，让堆替我们用 $O(\log n)$ 自动找出最小两堆：



- 若不用堆而每次找最小再重排：30000 轮每轮扫一遍约 4.5×10^8 次，勉强卡常或直接 **TLE**。
- 小根堆每次 top/pop/push 最多只调 $\log_2 30000 \approx 15$ 步，总操作不到 150 万次，几十毫秒稳过！

本页要点

模型映射极其直接：当前果子堆集合 = 小根堆 pq；“选最小两堆合完放回” = 两次 pop + 一次 push！

例 2 ｜合并果子｜样例 2 简化手推跟踪表

用带重复数字和合并反弹的 6 堆数据 [3，5，1，7，6，1] 演示（原理同样例 2）：

轮次	堆中当前元素（升序示意）	弹出最小两堆	新堆放回堆	累计 ans
初始	[1，1，3，5，6，7]	—	—	0
1	[1，1，3，5，6，7]	1，1	放回 2 → [2，3，5，6，7]	2
2	[2，3，5，6，7]	2，3	放回 5 → [5，5，6，7]	$2+5=7$
3	[5，5，6，7]	5，5	放回 10 → [6，7，10]	$7+10=17$
4	[6，7，10]	6，7	放回 13 → [10，13]	$17+13=30$
5	[10，13]	10，13	放回 23 → [23]	$30+23=53$
终止	[23]（size == 1）	size > 1 为假	循环结束	输出 53

本页要点

看第 3 轮：合出的 10 自动沉到 6，7 后面，第 4 轮优先合 6，7——这就是放回堆的威力！

例 2 ｜合并果子｜核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
1. 建小根堆并读入 n 堆	`pq = {1, 2, 9}`	是否写全了 `greater<long long>`?
2. 判 `pq.size() > 1`	$3 > 1$，进循环	若 $n = 1$，会不会错误进入循环?
3. 弹两堆累加 `a + b`	`1 + 2 = 3, ans = 3`	两个值是否都及时 `pop()` 了?
4. 把 `a + b` 推回堆	`pq = {3, 9}`	有没有漏掉 `pq.push(a + b)`?

相对模板改了哪里？

直接套用哈夫曼小根堆合并模板；特别把堆内元素和 `ans` 都设成 `long long`，防止更大数据下溢出！

本页要点

循环条件必须是 `pq.size() > 1`，这样 $n = 1$ 时直接跳过循环输出 0，既对又安全！

例 2 | 合并果子 | 核心代码（1/2）：读入与建小根堆

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    if (!(cin >> n)) return 0;
    priority_queue<long long, vector<long long>, greater<long long>> pq;
    for (int i = 0; i < n; ++i) { long long x; cin >> x; pq.push(x); }
```

本页要点

小根堆三个模板参数 < 类型，容器 vector，比较器 greater> 缺一不可！

例 2 | 合并果子 | 核心代码（2/2）：贪心合并循环

```
    long long ans = 0;
    while (pq.size() > 1) {
        long long a = pq.top(); pq.pop();          // 取出最小第一堆
        long long b = pq.top(); pq.pop();          // 取出最小第二堆
        ans += a + b;
        pq.push(a + b);                            // 合并成新的一堆塞回堆中
    }
    cout << ans << '\n';
    return 0;
}
```

本页要点

完整源码见 code/merge_fruit.cpp，已用样例 1 和样例 2 验证通过！

例 2 | 合并果子 | 易错与 3 组自测边界

高频易错（对比排雷）

- 默认写成了大根堆：漏写 `greater<long long>`，样例 1 会算出 23 而非 15（WA）。
- 忘记放回新堆：只做一遍排序累加，遇到 [3,4,5,6] 会算出 37 而非 36（WA）。

3 组自测边界与预期输出

1. 最小边界 $n = 1$：1 / 20000 $\Rightarrow$ 0（一次都不用合并，不进入循环）。
2. 原题样例 **1**：3 / 1 2 9 $\Rightarrow$ 15。
3. 原题样例 **2**：10 / 3 5 1 7 6 4 2 5 4 1 $\Rightarrow$ 120（验证多堆反复进出小根堆）。

本页要点

提交前一定用 $n = 1$ 测试是否输出 0 而不触发空堆弹出 RE！

例 3 | 医院设置（一本通 1338）[提高]：先读懂要求

大白话题意

有一棵 n 点的二叉树，点 u 住着 w_u 名居民，相邻点距离为 1。选一点建医院，使总路程 $\sum w_u \cdot d(u,s)$ 最小。

- 输入：第 1 行 n（$n \le 100$）；下接 n 行每行给 w_u、左链接 l、右链接 r（0 表示空）。
- 处理：居民能顺边上下穿行，枚举医院 $s = 1 \sim n$，用 BFS 算距 s 的路程和。
- 输出：一行一个整数，最小总路程。

原题样例

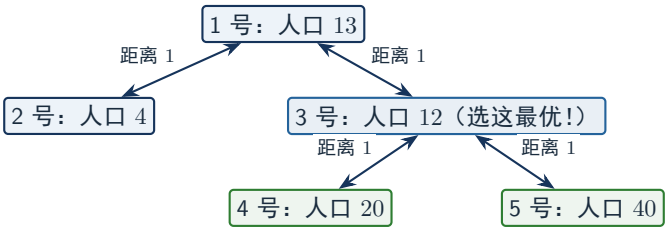
5 / 13 2 3 / 4 0 0 / 12 4 5 / 20 0 0 / 40 0 0 $\Rightarrow$ 输出 81

本页要点

居民能往上走也能往下走，绝不能当成有向树，必须建成无向图！

例 3 | 医院设置 | 思路与无向建树建模图解

把样例的 5 个点和父子边全部建成双向无向边，再用第 2 章的队列 BFS 就能一次求出其余点到医院 s 的距离：



- $n \le 100$，枚举 n 个候选点 s，每次用邻接表 BFS 求一遍距离只需 $O(n)$，总共 $O(n^2) \le 10^4$ 次操作，瞬间出解！

本页要点

当建在 3 号点时，人口最多的 20 和 40 只需走 1 步，而 2 号点需要穿过 1 号走 2 步到达 3 号！

例 3 | 医院设置 | 样例手推对比表

把医院分别建在 $s=1$ 与 $s=3$ 处，比较每个节点的距离 $d(u,s)$ 与路程贡献 $w_u \cdot d(u,s)$：

节点 u	人口 w_u	建在 $s=1$ 时距离	贡献 $w_u \cdot d(u,1)$	建在 $s=3$ 时距离	贡献 $w_u \cdot d(u,3)$
1	13	0	0	1	$13 \times 1 = 13$
2	4	1	$4 \times 1 = 4$	2	$4 \times 2 = 8$
3	12	1	$12 \times 1 = 12$	0	0
4	20	2	$20 \times 2 = 40$	1	$20 \times 1 = 20$
5	40	2	$40 \times 2 = 80$	1	$40 \times 1 = 40$
合计	89	—	**136**	—	**81**（最优）

本页要点

同理可手算 $s=2$ 得 193、$s=4$ 得 150、$s=5$ 得 130，故最小加权距离和为 81！

例 3 ｜医院设置｜核心步骤与相对模板改动

按顺序计算	纸上得到	核对问题
1. 读入 w_u 与非零孩子 l, r	连 4 条双向边	遇到 0 有没有跳过？
2. 枚举医院位置 $s = 1 \dots n$	依次设 $s = 1, 2, 3, 4, 5$	每次是否重置了 `dist`？
3. 从 s 做第 2 章队列 BFS	算出各点 `dist[u]`	判了 `dist[v] == -1` 吗？
4. 累加 `w[u]*dist[u]` 取最小	`best = min(136, 81) = 81`	`best` 初值是 `LLONG_MAX` 吗？

相对模板改了哪里？

第 1 节树上 BFS 只从 1 号根跑了一次；本题外层循环枚举所有点作根，累加带人口权重距离！

本页要点

枚举根 BFS 只需 $O(n^2)$，比 Floyd 更短更直观！

例 3 | 医院设置 | 核心代码（1/2）：读入人口与双向建图

```cpp
int n;
if (!(cin >> n)) return 0;
vector<long long> w(n + 1);
vector<vector<int>> g(n + 1);
for (int u = 1; u <= n; ++u) {
    int l, r; cin >> w[u] >> l >> r;
    for (int v : {l, r})
        if (v != 0) {                          // 跳过 0 号空儿子
            g[u].push_back(v);
            g[v].push_back(u);                 // 务必建双向边！
        }
}
```

本页要点

C++11 范围循环 `for (int v : {l, r})` 能同时简洁处理左、右两个儿子，避免写两遍重复连边！

例 3 | 医院设置 | 核心代码（2/2）：枚举根 BFS 求最小路程和

```
long long best = LLONG_MAX;
for (int s = 1; s <= n; ++s) {
    vector<int> dist(n + 1, -1); queue<int> q;
    dist[s] = 0; q.push(s);
    long long cost = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        cost += w[u] * dist[u];                    // 累加人口乘距离
        for (int v : g[u]) if (dist[v] == -1) { dist[v] = dist[u] + 1; q.push(v); }
    }
    best = min(best, cost);
}
cout << best << '\n';
```

本页要点

完整源码见 code/hospital_tree.cpp，已通过 C++17 编译与样例实测！

例 3 | 医院设置 | 易错与 3 组自测边界

高频易错（对比排雷）

- 只连了有向边 `u -> l`，`r`：医院设在 3 号点时，走不到 1 号和 2 号点，`dist` 算成 `-1` 导致全错（WA）。
- 没跳过 `0` 号空儿子：所有叶子都连向 0，树变成图且最短路绕道 0 号点（WA）。

3 组自测边界与预期输出

1. 单点树 $n = 1$：`1 / 100 0 0` $\Rightarrow$ `0`（居民全在医院所在点）。
2. 斜链重点偏右：`3 / 1 2 0 / 100 3 0 / 1 0 0` $\Rightarrow$ 建在 2 号得 `2`。
3. 原题样例：`5` 个节点如前 $\Rightarrow$ 输出 `81`。

本页要点

每次枚举新起点 s，务必重新初始化 `dist` 全为 `-1`，且 `best` 用 `LLONG_MAX`！

树、二叉树与堆高频错误对照与定位指南

题目 / 场景	常见错误写法	正确做法与原因
小根堆写反类型	写 `priority_queue<long long>`	STL 默认是大根堆，小根堆必带 `greater<>`
合并果子不放回	排一次序从头累加	漏掉新堆重排，遇 `[3,4,5,6]` 全 WA
医院设置连单向	只存 `u -> l, r`	儿子走不到兄弟，必须按无向图建双向边
先中序下标算错	右段写成 `pl + left_sz`	左子树起点 `pl+1`，右段起于 `pl+1+left_sz`
单点 $n = 1$ 特判	强行弹出两堆求合并	写 `while (pq.size() > 1)`，剩 1 堆自动输出 0

- 定位口诀：**WA** 查 `greater<>`、无向双向边、下标差 1；**RE** 查空堆调用与右儿子越界；**TLE** 查 `dist[v]==-1`。

本页要点

树上往来必建双向边；小根堆三个参数缺一不可！

提交前检查清单（六必查）

1. 需要每次取最小值的堆问题，是否明确写出了小根堆三个参数带 `greater<>()`？
2. 累加合并代价或树上加权路径总和时，是否已经把答案变量声明为 `long long`？
3. 二叉树遍历区间划分时，是否用一组 3 个节点的小树走了一遍 `pl+1+left_sz` 有没有多加或少减 1？
4. 树上两点互达问题（如《医院设置》），是否给父子之间连上了双向边，并在读到 0 时跳过空儿子？
5. 当 $n = 1$（只有一个节点/一堆果子）时，循环是否安全退出且不弹出空堆导致 RE？
6. 若用“先序 + 后序”，是否牢记没有中序是不能唯一确定二叉树的（除非已补齐空节点）？

最后一步

对单点 $n = 1$ 和左偏斜链（只有左儿子没有右儿子）各测一组再提交！

本章全部练习与完成标准

1. 学校 OJ：P1339 | 求后序遍历（[基础]，一本通 1339，code/postorder_tree.cpp）：先序定根 + 中序劈两半还原后序。
2. 学校 OJ：P1340 | 扩展二叉树（[基础]，一本通 1340）：补齐空结点 '.' 的先序唯一还原树，输出中序和后序。
3. 学校 OJ：P1365 | FBI 树（[基础]，一本通 1365）：把 2^N 长的 01 串从中间对半递归建树，输出后序遍历。
4. 学校 OJ：P1369 | 合并果子（[基础]，一本通 1369，code/merge_fruit.cpp）：小根堆每次取最小两堆合并放回（哈夫曼贪心）。
5. 学校 OJ：P1338 | 医院设置（[提高]，一本通 1338，code/hospital_tree.cpp）：二叉树建成无向图，枚举根做 BFS 求最小加权距离和。

完成标准

能手画先序 + 中序还原任何小二叉树；能一分钟写出小根堆上浮下沉与哈夫曼合并；独立 AC 上述 5 道习题。

课后安排与下一章预告

- 复习：合上代码再推一遍 `build(pl, pr, il, ir)` 的左右边界公式，并用 `abdec + dbeac` 验算一遍；手画一遍小根堆 `push` 和 `pop` 过程。
- 练习：独立复写 `code/` 下 3 份源码，并额外补完《扩展二叉树》与《FBI 树》，重点测试单点树与偏斜链表树。
- 下一章预告：第 4 章《并查集与 ST 表》——第 3 章我们自顶向下找左右孩子，下一章反过来用树“自底向上认总掌门”实现近乎 $O(1)$ 的集合合并判环，并学会用可重叠倍增块 $O(1)$ 秒答静态区间最值！

本章一句话

树用分支管层次，遍历把大树劈成左右两半，堆靠上浮下沉 $O(\log n)$ 守极值。

谢谢！