奇妙的二叉树

（王咏刚 2003 年 9 月）

1 问题引入

今天，我们要讨论的是数据结构和算法的设计问题。

“数据结构？算法？你没搞错吧？考完‘高程’的当天夜里，我就把广义表和有向图忘得一干二净了。现在的程序员，会用鼠标摆摆控件，会把控件和数据源绑在一起，就不愁没饭吃了，谁还会傻乎乎地抱着 Knuth 那三本大部头[①]不放呢？”

也许，你说的没错。现在，除了参加信息学“奥赛”的中学生，恐怕没多少人肯花时间去钻研数据结构和算法。不过就我所知，在真正的软件企业里，能够在程序设计中熟练运用数据结构与算法知识，仅凭三五行代码就把程序效率提高十倍的程序员仍然比那些只会拖放控件的程序员吃香，仍然可以拿到足以让同侪艳羡的薪金。因此，即便你没时间也没精力去发明新的算法，但从生计考虑，我还是建议大家把常用的算法与数据结构记熟，并懂得灵活运用的法门——本文讨论 Huffman 算法的目的也正在于此。

“什么什么？Huffman 算法？不就是用二叉树构造前缀编码的算法吗？不就是按照权值大小，每次选两个最小的结点拼成子树，如此反复并最终建成一棵二叉树的算法吗？这点儿东西我 N 年前就烂熟于胸了，还需要讨论吗？”

没错，Huffman 算法非常简单，每一个学过《数据结构》的学生都可以在半小时内编程实现。最直接的实现方法是用指针来维系树形结构，树中每个结点的定义如下：

```
typedef struct HTNode_T
{
  unsigned long weight; // 权值
  HTNode_T* parent; // 双亲结点指针
  HTNode_T* lchild; // 左子树指针
  HTNode_T* rchild; // 右子树指针
}HTNode, *HuffmanTree;
```

程序为每个待编码元素动态分配结点空间，每次选出**两个最小的结点，通过指针连成新的结点……具体的算法**很多教科书里都有，我就不多费口舌了。

不过，我必须在这里郑重指出，这种实现方法只是一种“最原始”的实现方法，它离真正实用的 Huffman 算法还相距甚远。在全文检索、专家系统、数据库、数据压缩、网络通信等真正需要 Huffman 编码的场合，Huffman 算法的实现远比上面这个算法精妙得多。

怎么样？如果你从未写过真正实用的 Huffman 算法，那不妨现在就开动脑筋，仔细推敲一下：我们该如何改进上面这种“最原始”的算法？其中共有多少可改进之处呢？

2 一些题外话

在讨论“实用”的 Huffman 算法之前，我们先来看另一个和“实用”相关的话题：

算法也罢，数据结构也罢，任何出色的设计都只有在最适宜的环境中才能发挥效力。

这句话的意思是说，算法和数据结构相当于程序的引擎，非常重要，但如果你把百分之百的精力都放在寻找最好的算法或数据结构上，而忽略了其他相关因素，那就好比拥有宝马引擎的威廉姆斯赛车舍米其林轮胎不用，只安装了四只英国乡下产的廉价车胎——这时，就算有八个小舒马赫外加十二个蒙托亚，威廉姆斯恐怕也没法和法拉利一决高下了。

仅就我们熟悉的排序算法而言：许多人知道快速排序算法的平均时间复杂度为 O(n log n)，是所有基于“比较/交换”的排序算法里平均效率最高的算法；一些细心的人也知道 Flashsort[②]等基于“分类”的算法可以凭借其 O(n) 的时间复杂度在速度上超越快速排序算法；更多的专业人士则清楚，在数据库索引、全文检索等应用领域，无论多快的内部排序算法都必须和多路归并等外部排序算法以及 B+树等文件结构相结合，才能满足海量数据管理的需要。

但是，很多人并不知道，如果想让排序速度达到极致，仅有上面这些出色的算法和数据结构还远远不够，我们必须考虑计算机硬件乃至整个运行环境对算法效率的影响。比方说，有大量数据需要排序时，内存中的排序主要占用 CPU 资源，而外部数据文件的归并则主要占用磁盘 I/O 资源，如果我们能让内部排序和外部排序并发运行，就可以更充分地利用系统资源，成倍地提高排序速度。再比方说，如果计算机配有多个 CPU，那就有必要将内存中的排序改变为多线程或多进程的模式，在多个 CPU 上同时运行，这一样可以让我们的排序算法事半功倍。

清华大学计算机系的一个小组在 2002 年 PennySort 世界排序大赛上胜出，并将原世界纪录提高了 1 倍左右，这是一件足以让所有中国程序员骄傲的事情。在比赛中，清华大学的参赛小组使用的正是一种基于多任务并综合考虑了 CPU、磁盘 I/O、最优并行方案、资源优化等多种因素的单机并行排序算法——THSORT[③]。该算法的出色表现说明，任何软件开发工作都是一项系统工程，都类似于参加一场 F1 大赛：如果我们不能像法拉利等优秀车队那样，在车手、引擎、空气动力学、轮胎、燃油、后勤保障、比赛策略等各方面都力求完美，恐怕就只能在赛道上被对手

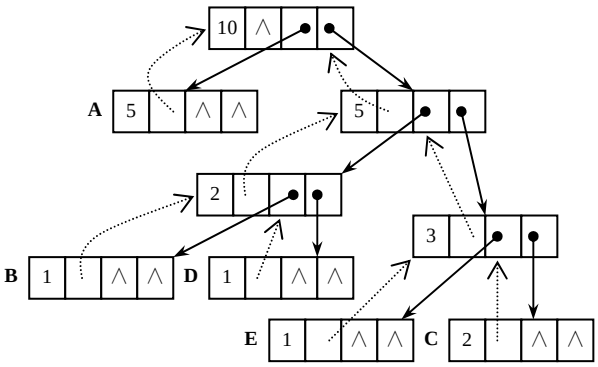
不断套圈，或是在赛程还未过半时就拖着浓烟退出比赛了。

3 案例分析

本文案例的目标是改进依靠指针来管理二叉树的“最原始”的 Huffman 算法。假设测试数据包含 5 个待编码元素，它们的权值分别为：

元素	A	B	C	D	E
权值	5	1	2	1	1

那么，“最原始”的 Huffman 算法将构建出下面这样一棵 Huffman 树：



这是一棵深度为 4 的严格（Strict）二叉树。如果左子树取 0，右子树取 1，则 5 个元素的编码如下：

元素	A	B	C	D	E
编码	0	100	111	101	110

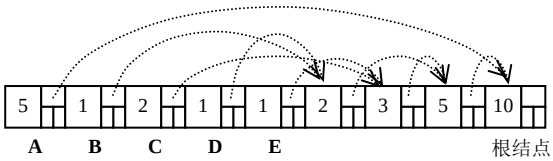
第一次改进

很显然，上面这种以指针管理二叉树的做法需要为每个结点动态分配存储空间，动态内存分配本身是一种极为耗时的操作。此外，为了能够在所有结点中找出最小的两个结点，我们还必须维护一个包含所有孤立子树指针的集合，这需要更多的存储空间。

既然许多《数据结构》教材已经指出，待编码元素数目为 n 的 Huffman 树共有 2n-1 个结点（这是严格二叉树的特性），我们不妨直接使用大小为 2n-1 的数组来管理二叉树，数组中每个单元的结构如下[④]：

```
typedef struct
{
 unsigned long weight; // 权值
 int parent, lchild, rchild; // 双亲和子树索引
}HTNode, *HuffmanTree;
```

这样，用数组的下标作为双亲或左、右子树的索引，并在大小为 2n-1 的数组中完成结点查找和二叉树的构建，就省去了动态分配、释放内存的时间和额外的存储开销。最终，我们可以得到下面这样的静态 Huffman 树：



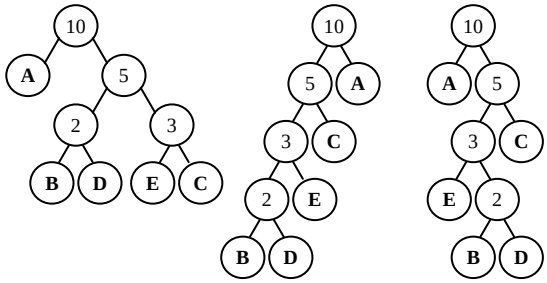
虚线表示指向双亲结点的索引。
这里略去了左右子树的索引。

仅从运行速度上讲，这种方法要比“最原始”的算法快 1 倍以上。此次改进说明：

为了提高运行效率，程序中要尽量避免频繁的内存分配和释放操作。

第二次改进

仔细观察不难发现，同样的一组待编码元素可能构建出许多棵不同的 Huffman 树。比方说，只要简单地调整最小结点的候选顺序，前面提到的 5 个元素就可能构建出深度和结构都不尽相同的 Huffman 树来：



不同的 Huffman 树将产生不同的元素编码表。即使在同一棵 Huffman 树上，元素的编码也取决于我们对左右子树编码的约定（左子树取 0 还是右子树取 0）。也就是说，同一组元素的 Huffman 编码可能有许多种，如果不作其他约定，我们就必须在编码的同时完整地保存元素编码表或是整棵 Huffman 树，以便将来能正确地解码，而这将消耗相当多的存储空间。

因此，在改进算法之前，为了规范 Huffman 编码的生成方式，我们需要熟悉一种前人给出的 Canonical Huffman 编码。这是一种规范的编码方式，其具体编码步骤如下[⑤]：

(1) 生成 Huffman 树，获得每个叶子结点的深度，也即每个待编码元素的码长；
(2) 将所有元素的码长按从大到小排序；
(3) 排序后，令第 1 个元素编码的每一位都为“0”；
(4) 下一个元素的码长如果和前一个元素码长相等，其编码就简单地等于前一个元素的编码加 1；如果码长比前一个元素的码长小 1，其编码就等于前一个元素的编码加 1 后再右移 1 位；如果码长小 2 则加 1 后右移 2 位……
(5) 依此类推，得到所有元素的编码。

例如，生成 Huffman 树后，假设前面提到的 5 个元素码长分别为：

元素	A	B	C	D	E
码长	1	3	3	3	3

那么，它们对应的 Canonical Huffman 编码为：

元素	A	B	C	D	E
编码	1	000	001	010	011

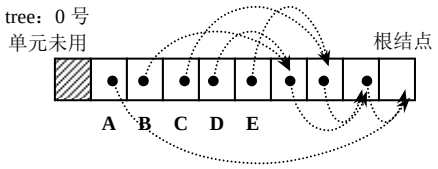
使用 Canonical Huffman 编码的好处是，只要保存了每个元素的码长，就可以在任何时候经计算得到唯一的前缀编码表。这不但可以省去大量的存储空间，而且可以将生成 Huffman 编码的算法简化为获得元素码长的算法。

考察第一次改进时得到的 Huffman 算法：数组中每个结点都必须保存双亲结点和左、右子树的索引，这是因为，生成 Huffman 编码时，我们必须知道每一次分支是向左还是向右，以便确定编码的每一位是 0 还是 1。如果使用 Canonical Huffman 编码，我们不但没必要存储每个结点的左、右子树索引，甚至也没有必要永久保存每个结点的权值——我们的目的仅仅是得到每个元素的码长。

简单地，我们可以用一个 unsigned long 型数组来解决所有问题：

```
unsigned long* tree = new unsigned long[2*n];
```

在生成编码之前，数组 tree[1]到 tree[n]存储了 n 个元素的权值。每次选出两个最小的结点后，我们就把这两个结点的权值替换为双亲结点的索引，把两个结点的权值之**和写入其双亲结点对应的单元……最终可以得到下面这样**一棵只包含双亲结点索引的静态 Huffman 树：



在这样的静态树中，令根结点码长为 0，并自右向左计算，每个结点的码长等于其双亲结点码长加 1，即可求出所有元素的码长，然后按照 Canonical Huffman 编码的规则生成 Huffman 编码即可。

因为采用了更简单的数据结构，这一次改进不但减少了存储空间的消耗，而且在运行速度上也比上一次改进快了将近 20%。此次改进说明：

对于问题本身的简化和规范（从普通 Huffman 到 Canonical Huffman）通常都可以起到简化算法与数据结构，并提高运行效率的作用。

第三次改进

还可以继续改进吗？别忘了，我们还没有讨论过，在构建 Huffman 树的过程中，到底该如何选出权值最小的两个结点。如果每次的选择操作都只是简单地在所有孤立子树中顺序搜索，那么我们的整个搜索过程就类似于简单选择排序，其时间复杂度必然是 $O(n^2)$。这对于待编码元素数量较大的场合（比如全文检索系统中的词表管理）就显得太慢太慢了。

显然，我们可以先用快速排序算法对数组中的所有元素按权值排序，然后依次选出两个权值最小的元素。

在这种改进方式中，有一个需要注意的问题：每选出两个结点，合并为一个子树后，新子树的权值（即两个结点权值之和）也需要重新插入到有序的权值序列中。我们可以在有序序列中简单地搜索新子树的插入位置，也可以用折半搜索来提高效率。

经过这样的改进，算法的运行效率能提高几十倍甚至数百倍（元素数量越多，速度优势就越明显）！此次改进说明：

除了应付考试以外，多背些教科书上的时间复杂度公式还是有好处的——我们至少可以知道，在数据量较大时，一定要避免与 $O(n^2)$ 这样的表达式打交道。

第四次改进

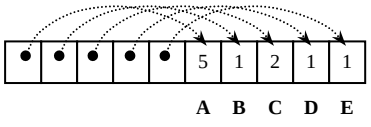
现在，我们已经有了规范的编码模型（Canonical Huffman），也使用了足够快的排序算法（快速排序），难道还有改进的余地吗？

考虑这样一个问题：生成 Huffman 树的算法要求我们每次选出权值最小的两个结点。我们真的有必要先把所有结点排序，并不断把新生成的结点插入到有序序列中吗？如果存在一种数据结构，可以把数据组织成一种理想的模型——既不用先对所有数据排序，也不需要穷举式的搜索，只凭少数几次比较和交换，就可以自动“弹”出最小的数据元素来——那岂不是更加完美吗？

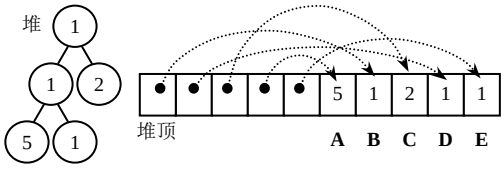
的确存在着这样一种速度奇快的选择算法，而且，它的原型还是我们在《数据结构》课程上再熟悉不过的一种通用算法——还记得堆排序算法（Heap Sort）吗？堆排序算法的主导思想是构造一棵满足堆的性质的完全二叉树，该二叉树的根结点总是所有元素中最小（或最大）的一个。而且，堆排序算法提供了一种重建堆的机制，能让新的数据元素在堆中迅速归位。

设想一下，如果我们在生成树之前，先把待编码的所有元素按权值大小组织成堆（其时间花费要远低于一次完全排序），选出堆顶元素；仿照堆排序算法，把剩下的元素重建成堆，再选出堆顶元素；然后把已选出的两个最小结点合并为新的结点，把新结点的权值放到堆中，完成堆的**重建……如此反复，不就可以在一个类似堆排序的过程里**完成整个 Huffman 树的构建吗？

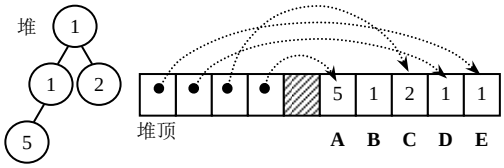
例如，对于前面 A 到 E 共 5 个待编码元素，我们使用大小为 10 的 unsigned long 数组，先将 5 个元素放到数组的后半部分，把数组的前半部分视为指针，依次指向每一个待编码元素：



只移动指针，让数组的前半部分构成一棵满足堆的特性的完全二叉树，第 0 号元素是堆顶元素：

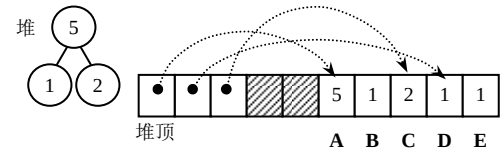


取出堆顶元素，将最后一个元素移到堆顶：

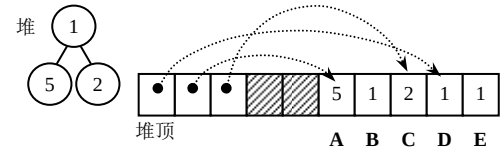


让堆顶元素自上而下比较和交换，找到合适位置，完成堆的重建。在本例中，堆顶元素不用移动就已经满足堆的特性了。

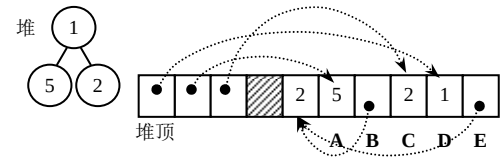
再次取出堆顶元素，将最后一个元素移到堆顶：



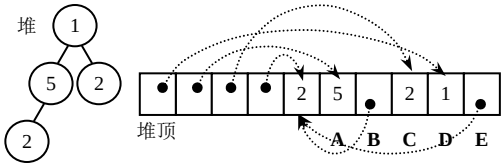
堆顶元素向下移动，完成堆的重建：



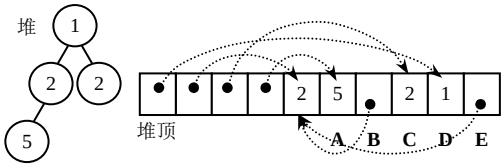
把已经取出的两个元素（权值均为 1）合并，得权值为 2 的子树，放在最后一个空闲单元上，并将两个元素的权值改为指向该单元的索引：



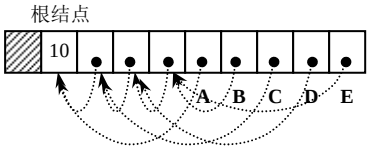
将新生成的权值为 2 的结点放到堆尾：



再次完成堆的重建：



反复执行上述操作。最终，堆中所有元素都被取出，数组内生成了一棵真正的 Huffman 树：除了 0 号单元和根结点（1 号单元）外，每个数组单元中存储的都是指向其双亲结点的索引：



接下来的事情就非常简单了，我们可以从根结点出发计算所有叶子结点的码长，然后使用 Canonical Huffman 规则获得所有元素的编码。

因为简化了对所有结点排序以及将新结点插入到有序序列中的算法，这种根据堆排序原理创建的算法在效率上又前进了一大步，在元素数量较多时，速度可提高 5～10 倍。此次改进说明：

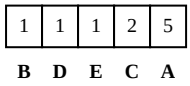
记住每一种算法和数据结构的实现方法还远远不够，学会在编程实践中灵活运用（比如我们对堆排序算法的灵活运用）才是我们的终极目标。

第五次改进

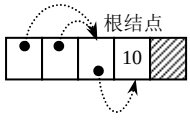
上面类似堆排序的算法已经算得上一种“实用”的 Huffman 算法了。但在某些时候，我们还可以继续改进。

例如，在索引管理、词典管理等特殊场合，待编码元素可能已经按权值顺序从小到大排列了。在这种情况下，Jyrki Katajainen 和 Alistair Moffat 为我们提供了一种空间复杂度最小的 Huffman 算法[®]。这种算法的神奇之处在于，我们可以在大小为 n 的数组中完成 n 个元素的 Huffman 树创建以及元素码长的计算工作。

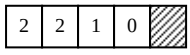
仍以前述 A 到 E 共 5 个元素为例，假设 5 个元素已按权值大小排好了顺序：



我们就在这个数组中，从左至右对所有元素做第 1 次遍历，两个两个地合并权值最小的结点，生成 Huffman 树。遍历的结果是：数组中 0 号到 n-3 号单元对应于 Huffman 树中除根结点和叶子结点外的所有分支结点，单元值为该分支结点的双亲结点索引；n-2 号单元对应于根结点，其中的数值是所有元素权值的总和。如下图：



然后，从右至左对数组做第 2 次遍历，自根结点起，设置所有分支结点的深度。第 2 次遍历的结果是：



这表明，在我们生成的 Huffman 树中，有 1 个根结点，1 个深度为 1 的分支结点和 2 个深度为 2 的分支结点。那么，该如何求得所有叶子结点的深度，也就是所有待编码元素的码长呢？

回忆一下严格二叉树的性质：树中没有度为 1 的结点，

每一层的叶子结点数量等于上一层的分支结点数量乘 2 后减去本层的分支结点数量。我们可以利用这一性质从上至下依次求得每层的叶子结点数量。别忘了，生成 Huffman 树之前，所有元素是按权值从小到大排列的，相应地，元素的码长则一定是按从大到小的顺序排列的。这样，我们就有可能在求得每层叶子结点数量的同时，把该层的深度作为码长写入到数组的相应位置中。

在上面的例子里，根结点下只有一个深度为 1 的分支结点，这意味着另一个子结点必然是一个深度为 1 的叶子结点，而这个叶子结点必然对应于权值最大也就是最右边的那个元素。

因此，我们的第 3 次遍历依然从根出发，自右向左计算每一层的叶子结点数量，同时从数组右边起依次为每个元素赋予正确的码长。第 3 次遍历结束后，数组中正好包含了所有元素的码长信息：

3	3	3	3	1

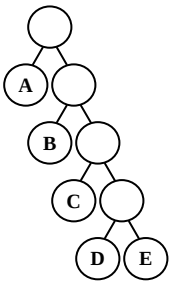
这一改进不但节省了一半存储空间，而且在元素已经有序的情况下，还要比前面的算法更快一些。此次改进说明：

数据结构和算法的发展永无止境，我们必须时常关注他人的研究成果和实践经验。

第六次改进

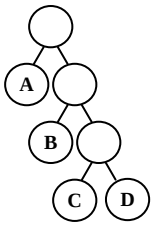
与 Huffman 算法有关的另一种改进是：在元素数目较多或权值差距较大时，元素的码长可能非常大。例如，当 50 个元素的权值分别为 2^0、2^1、2^2、2^3……2^{49} 时，前 2 个元素的码长将达 49 位。在这种情况下，为了方便程序对元素码长和编码的管理，常常要对元素码长进行限制，即规定元素的最大码长不能超过某个数值，如果有超长码长存在，就必须对 Huffman 树进行调整。

假设规定的最大码长为 3，A 到 E 共 5 个元素恰好构建出下面这样一棵 Huffman 树：

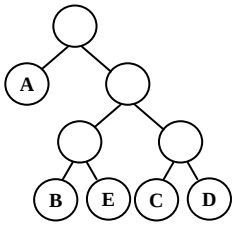


显然，元素 D 和 E 的码长为 4，超出了最大码长限制，我们需要根据二叉树的性质对元素位置进行调整。

首先，将超长的元素 D、E 连同它们的双亲结点从树中移去。这时，Huffman 树的第 3 层留下了一个空位，我们可以把元素 D 连在空位上：



从树的第 2 层开始逐层向上搜索，找 1 个叶子结点，在本例中是叶子结点 B。用一个分支结点替换 B，然后将 B 和 E 连入该分支结点：



现在，所有叶子结点的深度都不超过 3，我们得到了一棵符合要求的二叉树。尽管具体的实现要比这里给出的描述复杂一些，但此次改进仍然说明：

数据结构和算法的基础知识（例如二叉树的基本性质）即便是在复杂的环境下也一样能发挥极为重要的作用。

4 补充说明

因篇幅原因，前面对每种算法的描述可能不够完整。没关系，源代码可以说明一切。大家可以到 http://www.contextfree.net/wangyg/huffman/ 下载文中每一种算法的源代码——顺便提一句，在数据结构和算法领域，多读经典著作，多读他人的源代码，这要比自己抓耳挠腮、冥思苦想有效得多。

5 总结一下

- 千万不要放弃对数据结构和算法知识的学习。
- 关键是能否在具体环境中灵活运用数据结构和算法知识。
- 最基本的数据结构和算法知识往往也是最有价值的知识。

---

① Knuth D E. The Art of Computer Programming. Reading, Mass: Addison-Wesley, 1997

② Neubert K D. The Flashsort Algorithm. Dr. Dobb's Journal, 1998, 23(2):123~129

③ 施遥，张力，刘鹏. THSORT：单机并行排序算法. 软件学报. 2003, 14(2):159~165

④ 严蔚敏，吴伟民. 数据结构（C 语言版）. 北京：清华大学出版社, 1997

⑤ Schindler M. Practical Huffman coding. http://www.compressconsult.com/, 2003

⑥ Katajainen J, Moffat A. In-Place Calculation of Minimum-Redundancy Codes. http://www.cs.mu.oz.au/~alistair/, 2003