

21 | 哈夫曼 (Huffman) 树：将数据压缩后再传输更省带宽

2023-03-31 王健伟 来自北京

《快速上手C++数据结构与算法》



你好，我是王健伟。

前面我们已经讲过了很多种二叉树，这节我想再和你分享一种特殊的二叉树——哈夫曼树 (Huffman Tree) 。

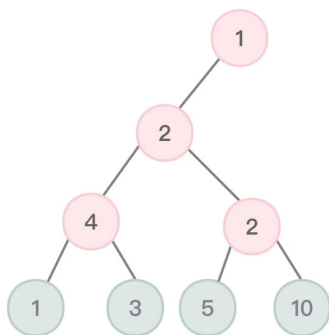
哈夫曼树也有人称为**霍夫曼树**或**最优二叉树**。先说点有趣的，哈夫曼 (David Huffman) 是美国的一位数学家。他在 1952 年发明了哈夫曼编码（一种二进制编码），该编码中用到了一种特殊的二叉树，人们为了纪念他的成就，将所用到的特殊二叉树称为哈夫曼树。

当然，知道这个可不算真正了解哈夫曼树，在了解哈夫曼树之前，首先要学习几个基本概念。

基本术语、概念

首先，**节点的权**。它指的是给树中的各个节点一个数值，用来表示某种现实含义。比如用 1-10 之间的数字表示某个节点的重要性或者表示该节点出现的频率等，这个数字就叫做节点的权（权值）。

如图 1 所示：

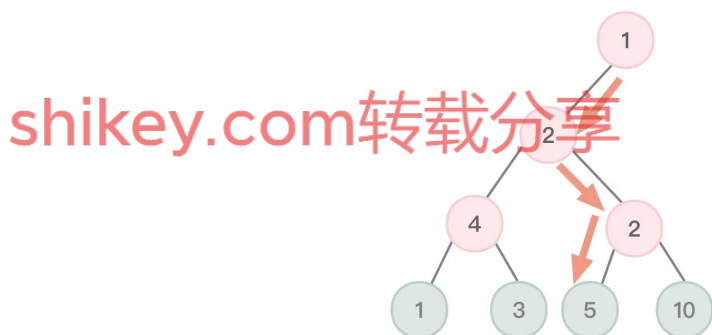


极客时间

图1 一棵二叉树，节点中的数字表示节点的权

再来，什么是**从根节点到某节点的路径长度**？这个指的是从根节点到该节点所经过的路径段数。比如对于权值为 5 的节点（是个叶子节点），从根节点到该节点的路径长度为 3。换句话说，从权值为 5 的节点向根节点回溯要经过 3 个前辈节点，所以从根节点到权值为 5 的节点的路径长度为 3。

如图 2 所示：



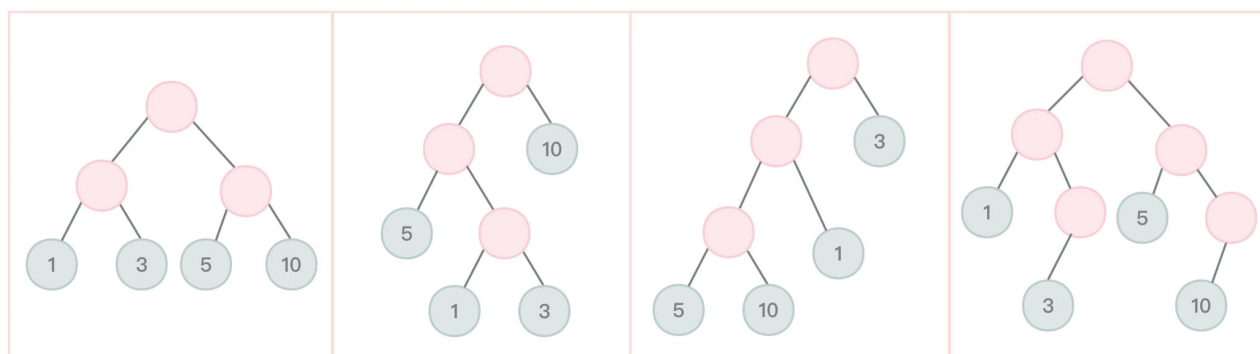
极客时间

图2 从根节点到权值为5的节点的路径长度为3

接下来就是 2 个重要概念的铺垫了。

一个是**节点的带权路径长度**。它的英文是 Weighted Path Length (缩写 WPL) 。节点的带权路径长度表示从根节点到该节点的路径长度与该节点权值的乘积。比如对于权值为 5 的叶子节点，从根节点到该节点的路径长度为 3，所以该节点的带权路径长度为 $3 \times 5 = 15$ 。

另一个是**树的带权路径长度**。树的带权路径长度是树中所有叶子节点的带权路径长度之和。图 2 这棵树的带权路径长度是 $(1 \times 3) + (3 \times 3) + (5 \times 3) + (10 \times 3) = 57$ 。



极客时间

图3 计算一下图中的4棵二叉树的带权路径长度

看一看图 3 中的 4 棵树的带权路径长度，在图 3 中：

第 1 棵树的 WPL 为 $(1 \times 2) + (3 \times 2) + (5 \times 2) + (10 \times 2) = 38$

第 2 棵树的 WPL 为 $(5 \times 2) + (1 \times 3) + (3 \times 3) + (10 \times 1) = 32$

第 3 棵树的 WPL 为 $(5 \times 3) + (10 \times 3) + (1 \times 2) + (3 \times 1) = 50$

第 4 棵树的 WPL 为 $(1 \times 2) + (3 \times 3) + (5 \times 2) + (10 \times 3) = 51$

基础概念了解之后，我们来看看它们和哈夫曼树的关系。通过前面的计算发现，图 3 中的第 2 棵树的 WPL 值最小。无论是什么形状的二叉树，只要叶子节点是 1、3、5、10，那么 WPL 值最小也不会低于 32。所以 WPL 值为 32 的这棵二叉树就是哈夫曼树（带权路径长度最小）。

现在，我们可以给出哈夫曼树的正式定义：在含有 n 个带权叶节点的二叉树中，其中带权路径长度（WPL）最小的二叉树称为哈夫曼树，也称为**最优二叉树**。当然，哈夫曼树也不唯一，比如图 3 的第 2 棵树是一棵哈夫曼树，假如把该树的权值为 1 和 3 的节点互换后得到的二叉树 WPL 依旧为 32，也是一棵哈夫曼树。所以包含相同叶子节点的哈夫曼树也许会有多种形态。

哈夫曼树的构造及相关的代码实现

如果给定 n 个叶子节点，如何构造包含这 n 个叶子节点的哈夫曼树呢？我们说下几个步骤。

将这 n 个节点分别作为 n 棵仅含一个节点（根节点）的二叉树，这样就构成了一个二叉树集合 F 。

构造一个新节点，在集合 F 中选取两棵根节点权值最小的树作为这个新节点的左右子树，这里谁左谁右顺序任意，以此构造一棵新二叉树。这个新节点的权值是左右两棵子树根节点的权值之和。

从集合 F 中删除刚刚选取的两棵树，并将新得到的树加入到集合 F 中。

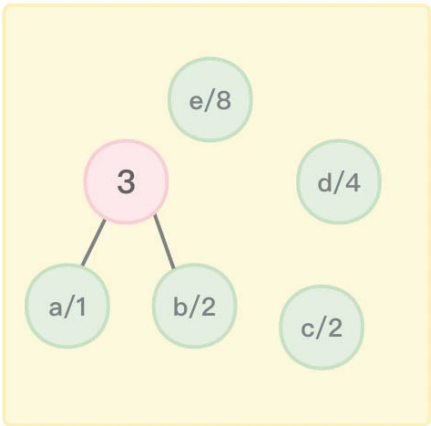
重复上面的 2 个步骤，直到集合 F 中只剩下一棵树为止，这棵树就是哈夫曼树。

我们通过图片理解一下整个步骤。如图 4，有权值分别为 1、2、2、4、8 的节点 a 、 b 、 c 、 d 、 e ，构成了一个二叉树集合 F ：



图4 5棵仅含一个节点的二叉树构成的二叉树集合 F

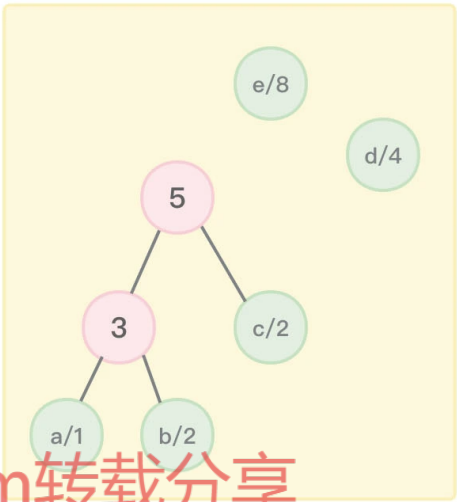
在集合 F 中，选取两棵根节点权值最小的树 a 和 b，当然，a 和 c 也行，因为 b 和 c 权值相同。然后构造一棵新二叉树，新二叉树根节点的权值是 $1+2=3$ ，如图 5 所示：



极客时间

图5 2棵权值最小的二叉树a、b构造一棵根的权值为3的新二叉树，目前二叉树集合F中只有4棵树

接着选取根节点权值为 2 和 3 的二叉树构造一棵新二叉树，其根节点的权值是 $2+3=5$ ，如图 6 所示：

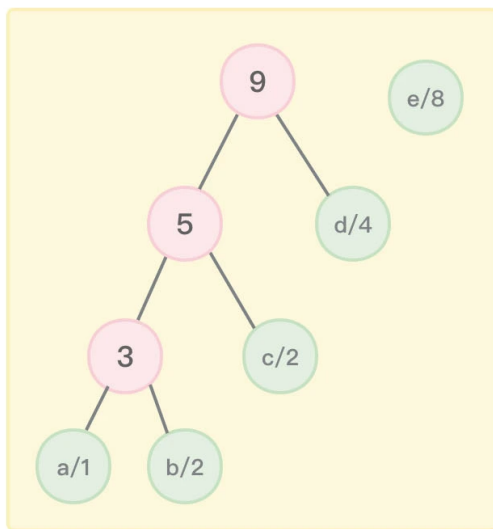


shickey.com转载分享

极客时间

图6 2棵权值分别为2和3的二叉树构造一棵根权值为5的新二叉树，目前二叉树集合F中只有3棵树

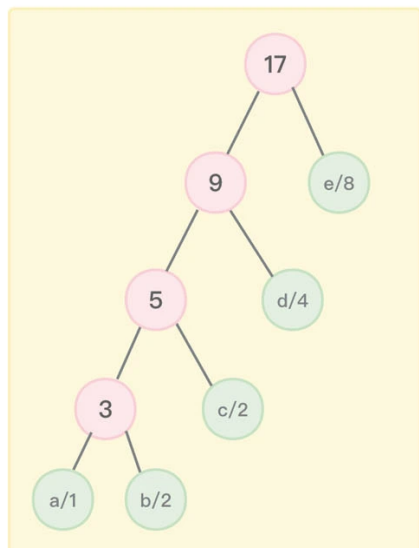
接着选取根节点权值为 4 和 5 的二叉树构造一棵新二叉树，其根节点的权值是 $4+5=9$ ，如图 7 所示：



极客时间

图7 2棵权值分别为4和5的二叉树构造一棵根权值为9的新二叉树，目前二叉树集合F中只有2棵树

接着选取根节点权值为 8 和 9 的树构造一棵新二叉树，其根节点的权值是 $8+9=17$ ，如图 8 所示：



极客时间

图8 2棵权值分别为8和9的二叉树构造一棵根权值为17的新二叉树，目前二叉树集合F中只有1棵树

目前集合 F 中只剩下一棵树了，所以这棵树就是哈夫曼树。这棵哈夫曼树的 WPL 为 $(8*1)+(4*2)+(2*3)+(2*4)+(1*4)=34$ 。

观察图 8，我们不难发现哈夫曼树的一些特性。

初始给出的节点都是哈夫曼树的叶子节点。

权值越大的叶子节点（比如节点 e）到根节点的路径长度越小，权值越小的叶子节点（比如节点 a、b）到根节点的路径长度越大。


因为有 n 个叶子节点，并且一共进行了 $n-1$ 次的两两合并最终得到哈夫曼树，而每合并一次多出一个节点（ $n-1$ 次合并就会多出 $n-1$ 个节点），所以哈夫曼树的节点总数是 $n+(n-1)=2n-1$ ，这个数据在编程的时候会用到。

哈夫曼树中没有度为 1 的节点。注意这里，节点拥有的子树数目，叫做节点的度。因为构造哈夫曼树的过程中，每个新节点都是有左右子树的。

哈夫曼树的子树仍旧是哈夫曼树。


哈夫曼树并不是唯一的，但相同叶子节点的哈夫曼树的 WPL 值一定是相等的。

哈夫曼树的代码实现并不复杂，因为给定叶子节点权值以及叶子节点数量后，整个哈夫曼树的节点数量是可以确定的（ $2n-1$ ），因此，可以创建动态数组来保存哈夫曼树。你可以参考下面的代码。

 复制代码

```
1 //哈夫曼树的节点
2 struct HFMTreeNode
3 {
4     int weight; //权值
5     int parent; //父亲（数组下标值）
6     int lchild; //左孩子（数组下标值）
7     int rchild; //右孩子（数组下标值）
8 };
```

shikey.com转载分享

 复制代码

```
1 //哈夫曼树：用一个数组来保存哈夫曼树
2 class HFMTTree
3 {
4 public:
5     HFMTTree(int nodecount, int* pweight) //构造函数
6         //参数nodecount代表要创建的哈夫曼树的叶子节点的数量
7         //pWeight代表叶子节点的权重数组
8     {
9         m_length = nodecount;
10        int iMaxNodeCount = 2 * m_length - 1;
```

```

11     m_data = new HFMTreeNode[iMaxNodeCount]; //哈夫曼树的节点总数是2n-1 (n代表哈夫曼树
12
13     for (int i = 0; i < iMaxNodeCount; ++i)
14     {
15         m_data[i].parent = -1; //-1标记未被使用
16         m_data[i].lchild = -1;
17         m_data[i].rchild = -1;
18     }
19     for (int i = 0; i < m_length; ++i)
20     {
21         m_data[i].weight = pweight[i];
22     }
23 }
24 ~HFMTree() //析构函数
25 {
26     delete [] m_data;
27 }
28
29 public:
30     //真正的开始创建哈夫曼树
31     void CreateHFMTree()
32     {
33         int idx1 = 0;
34         int idx2 = 0;
35
36         int iMaxNodeCount = 2 * m_length - 1; //2n-1是整个哈夫曼树的节点数量
37         int initlength = m_length;
38         for (int i = initlength; i < iMaxNodeCount; ++i)
39         {
40             SelectTwoMinValue(idx1, idx2);
41             m_data[i].weight = m_data[idx1].weight + m_data[idx2].weight; //新节点的权值等
42             m_data[i].lchild = idx1;
43             m_data[i].rchild = idx2;
44
45             m_data[idx1].parent = i;
46             m_data[idx2].parent = i;
47
48             m_length++; //SelectTwoMinValue()函数要用到该值
49         } //end for i
50         return;
51     }
52
53     //前序遍历二叉树(根左右)
54     void preOrder(int idx)
55     {
56         if (idx != -1)
57         {
58             cout << m_data[idx].weight << " ";
59             preOrder(m_data[idx].lchild);

```




```

60     preOrder(m_data[idx].rchild);
61 }
62 }
63
64 //获取树中节点数量
65 int GetLength()
66 {
67     return m_length;
68 }
69
70 private:
71 //选择两个根权重最小的节点，通过引用返回这个节点所在的下标
72 void SelectTwoMinValue(int& rtnIdx1, int& rtnIdx2)
73 {
74     int minval1 = INT_MAX;
75     int minval2 = INT_MAX;
76
77     //找最小值
78     for (int i = 0; i < m_length; ++i)
79     {
80         if (m_data[i].parent == -1) //父标记未被使用
81         {
82             if (minval1 > m_data[i].weight)
83             {
84                 minval1 = m_data[i].weight; //记录最小值
85                 rtnIdx1 = i; //记录下标
86             }
87         }
88     } //end for i
89
90     //找第二个最小的值
91     for (int i = 0; i < m_length; ++i)
92     {
93         if (m_data[i].parent == -1 && i != rtnIdx1) //注意&&后的条件，目的是把第一个找到
94         {
95             if (minval2 > m_data[i].weight)
96             {
97                 minval2 = m_data[i].weight; //记录最小值
98                 rtnIdx2 = i; //记录下标
99             }
100         }
101     } //end for i
102     return;
103 }
104 private:
105 HFMTTreeNode* m_data;
106 int m_length; //记录当前树有多少个节点【数组中多少个节点被使用了】
107 };

```

在 main 主函数中，增加下面的测试代码。

 复制代码

```
1 int weighLst[] = { 1,2,2,4,8}; //权值列表（数组），数组中的数据顺序无所谓
2 HFMTree hfmtobj(
3     sizeof(weighLst) / sizeof(weighLst[0]), //权值列表中元素个数
4     weighLst //权值列表首地址
5 );
6
7 hfmtobj.CreateHFMTree(); //创建哈夫曼树
8 hfmtobj.preOrder(hfmtobj.GetLength()-1); //遍历哈夫曼树，参数其实就是根节点的下标（数组最
```

执行结果如下。这个结果是对生成的哈夫曼树进行前序遍历（根左右）的结果。

17 8 9 4 5 2 3 1 2

通过对上述代码的阅读和分析不难发现，利用 1、2、2、4、8 这 5 个权值作为叶子节点创建的哈夫曼树，用数组保存后，数组内容应该如图 9 所示。有兴趣的话，你也可以增加一些代码来输出数组中的内容方便观察，当然通过跟踪调试查看也是可以的。



图9 用权值1、2、2、4、8创建的哈夫曼树在内存中以数组形式保存时的数据内容

把图 9 从上到下排列观察可能更加明显，如图 10 所示：

下标	权值 weight	父下标 parent		
[0]	1	5	-1	-1
[1]	2	5	-1	-1
[2]	2	6	-1	-1
[3]	4	7	-1	-1
[4]	8	8	-1	-1
[5]	3	6	0	1
[6]	5	7	2	5
[7]	9	8	3	6
[8]	17	-1	4	7

图10 用权值1、2、2、4、8创建的哈夫曼树在内存中以数组形式保存时的数据内容（竖着排列）

根据图 9 或图 10，也很容易知道哈夫曼树的构造过程直至最后得到所需的哈夫曼树，结果如图 11 所示：

shikey.com转载分享

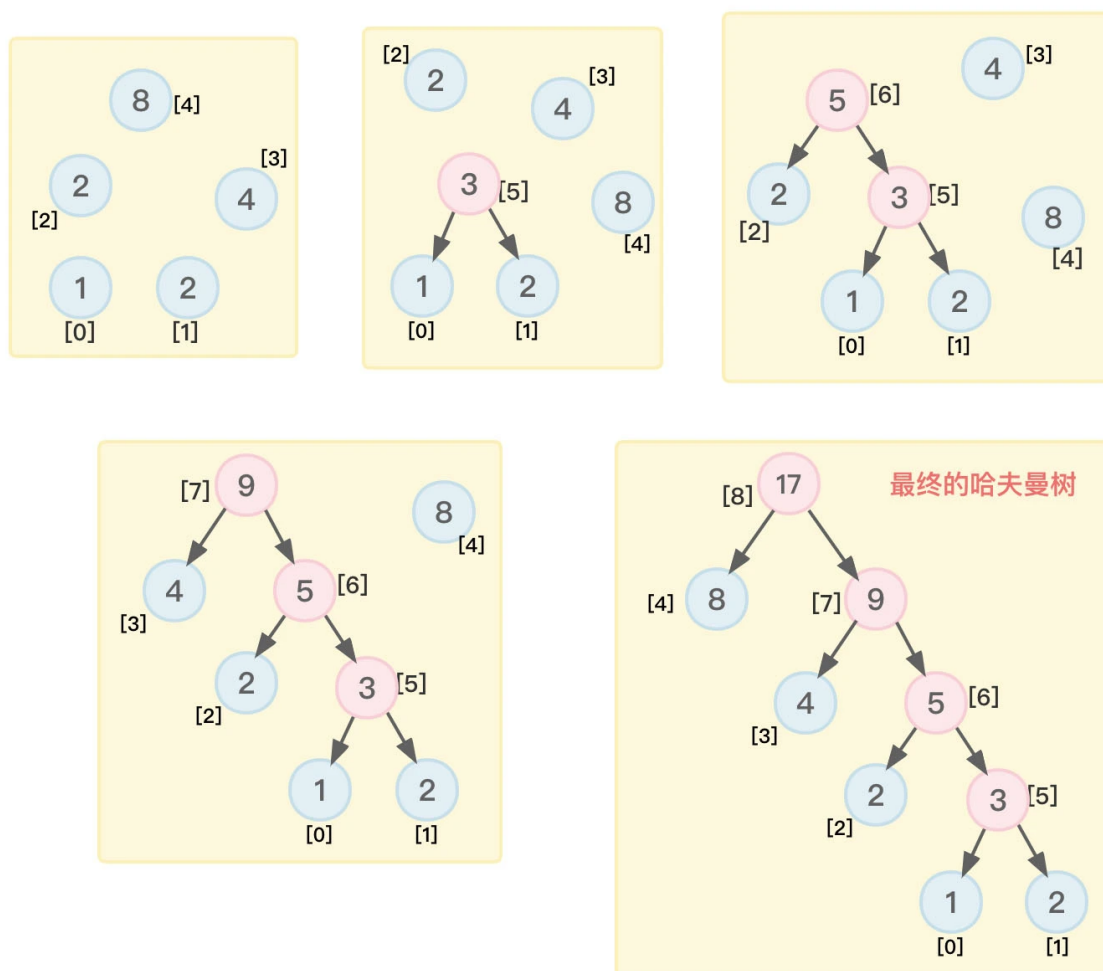


图11 用权值1、2、2、4、8创建哈夫曼树的过程展示

利用前序遍历（根左右）对图 11 的最后一幅图中显示的树进行遍历，得到的结果（权值）的顺序正是 17、8、9、4、5、2、3、1、2。

哈夫曼编码及相关的代码实现

哈夫曼树有什么用途呢——和我们最开始说的故事一样，用来创建哈夫曼编码（Huffman Coding——一种二进制编码）。哈夫曼编码是一种可以用于数据压缩的编码方式，比如你可以想象在 winrar 或 winzip 等压缩软件中采用这种压缩方式。而哈夫曼编码的构造过程是需要用到哈夫曼树的。

哈夫曼编码主要解决通信双方传输信息时针对信息压缩问题，通过传输最少量的信息来表达一段相同的内容。

设想有一段文字内容“AFDBCFBDEFDF”要通过网络传给别人。一般来说，传输一段信息时，往往采用二进制的 0 和 1（分别代表两种信号）来进行信息传输最便捷，所以考虑把要传输的这段文字内容进行编码。因为这段文字内容只涉及了 A、B、C、D、E、F 共 6 个字母。而用 3 位二进制数表示（编码）一个字母，则足可以表示 8 个字母（从二进制的 000 到二进制的 111），所以用 3 位二进制数（字符）表达这段文字内容涉及的 6 个字母绰绰有余，如图 12 所示，注意，这属于**固定长度编码**：

字母	A	B	C	D	E	F
二进制字符	000	001	010	011	100	101



图12 用3位二进制数字表示A、B、C、D、E、F共6个字母

在传输文字内容“AFDBCFBDEFDF”时，传输的数据就是编码后的“000101011001010101001011100101011101”。接收方可以按照双方事先的约定，也就是 3 位一划分来将二进制编码译码还原为真正的文字内容。但是如果传输的文字内容特别多，那么编码后的内容也将非常的长，这意味着传输的内容也会非常多。

事实上，在真正的数据传输中，不管传输的是英文字母、汉字等等，字母或者汉字出现的频率并不相同，比如在英文的 26 个字母中“E、A、T、I、N、O”出现的频率就会明显高于其他字母，而在汉字中“有、人、的、工、在、一、是”等出现的频率也会比其他汉字更高。

针对前面传输的文字内容“AFDBCFBDEFDF”中所包含的 6 个字母，可以粗略估算也可以假设它们出现的频率，按照出现频率一共 100% 计算，大略估计这 6 个字母的出现频率为 A：12%、B：15%、C：9%、D：24%、E：8%、F：32%。有了这种估计，就可以按照哈夫曼树来重新进行编码规划——将 A、B、C、D、E、F 这 6 个字母分别看做叶子节点，将这 6 个字母的百分比（去掉百分号）12、15、9、24、8、32 分别看做叶子节点的权值，这样就可以构造出一棵哈夫曼树。

如图 13 所示：

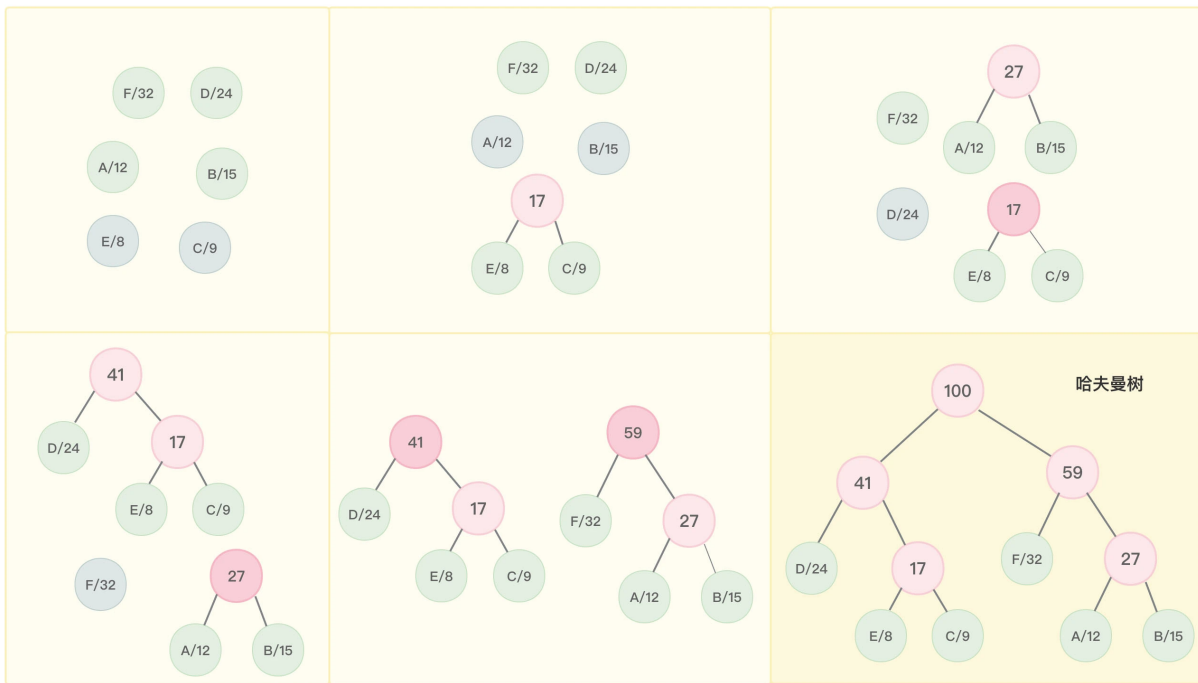


图13 以A、B、C、D、E、F作为叶子以一定的权值构造一棵哈夫曼树

针对图 13 所展示的哈夫曼树，如果左分支路径中的各个段用 0 标示，右分支路径的各个段用 1 标示，换句话说，从根节点出发，向左走表示二进制的 0，向右走表示二进制 1，那么这种用二进制字符表示字母的方案就可以映射为树的表示形式。如图 14：

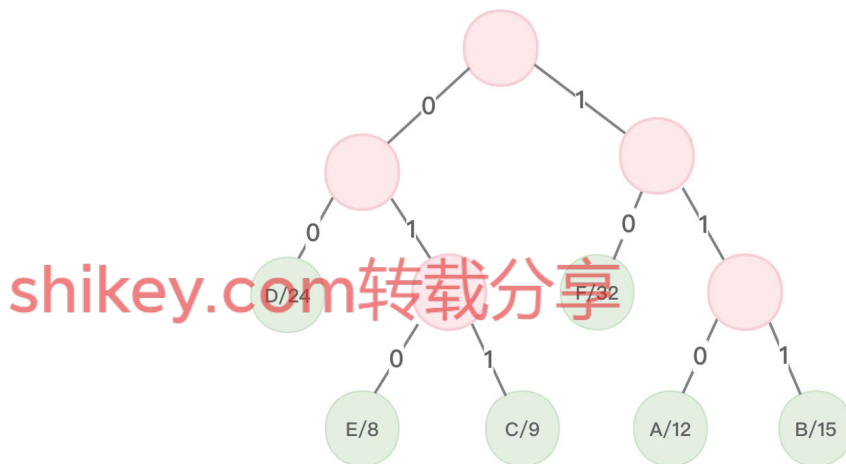


图14 将以A、B、C、D、E、F为叶节点的哈夫曼树左分支各段标示为0，右分支各段标示为1

从图 14 可以看到，从根节点出发，访问节点 D 需要经过的路径所包含的二进制数为 00，节点 E 经过的路径所包含的二进制数为 010.....，这意味着 D 的编码是 00，E 的编码是 010.....。那么哈夫曼树的**叶子节点**所对应的二进制编码如图 15 所示（这些字母对应的二进制字符编码就是**哈夫曼编码**）。

字母	A	B	C	D	E	F
二进制字符	110	111	011	00	010	10

极客时间

图15 用哈夫曼树对A、B、C、D、E、F重新进行二进制编码（哈夫曼编码）

从图 15 中可以看到，**出现频率最高的字母，尽可能用最短的编码以节省数据通信量**，所以这是一种**可变长度编码**，也就是不同的字母编码后对应的二进制字符长度不同。最终，要传输的文字内容是 “AFDBCFBDEFDF” ，而实际传输的内容是编码后的 “11010001110111011100010100010” 。可以看到，与原来需要传输的二进制字符对比一下。

原二进制字符串：000101011001010101001011100101011101 （36 个字符）

新二进制字符串：11010001110111011100010100010 （29 个字符）

这意味着需要传输的数据变少了，也就是数据被压缩了。节省了大概 19% 的保存或传输成本。显然，如果要传输的文字内容更多，所节省的成本也将更加可观。

那么如何从新的二进制字符串中把真正的文字内容解码出来呢？因为编码中只有 0 和 1，而且是一种可变长度的编码，在解码时其实是容易因混淆而导致解码错误的，所以，对于可变长度的编码，设计时必须保证**任何一个字母的编码都不可以是另一个字母编码的前缀**。比如字母 F 的二进制编码是 10，那么其他字母在编码时绝对不可以以 10 开头，观察图 15，字母 A、B、C、D、E 的二进制字符都不是以 10 开头。

这里涉及一个概念——**前缀编码**：如果在一个编码方案中，任何一个编码都不是其他任何编码的前缀（最左子串），则称该编码是前缀编码。哈夫曼编码就属于一种前缀编码。

举个例子，假设字母 C 的二进制编码不是 011 而是 101，因为以 10 开头了是不被允许的，那么如果传输的内容是 10110110，接收方在解码时可能会解码为 CCF，也可能会解码为 FAA，这样就产生了歧义和混乱。而按照图 15 这样编码，在收到新二进制字符串时，解码出来的内容只能是“AFDBCFBDEFDF”，绝不会解码成其他内容。当然，为了保证发送方发送的内容接收方能够成功解码，接收方手中也必须有一份图 15 所示的编码表。

好了，我们总结一下，哈夫曼编码是用构造哈夫曼树的方法来确定字符集的一种编码方案，属于一种前缀编码，在解码时可以保证解出的内容的唯一性。

字符集中的每一个字符作为叶子节点，将各个字符出现的频率作为该节点的权值，构造出哈夫曼树。

将哈夫曼树左分支路径中的各个段用 0 标示，右分支路径中的各个段用 1 标示。当然，左分支用 1 标示，右分支用 0 标示也可以，只要通信双方在编码和解码时做好一致的约定就可以。

从哈夫曼树的根节点到叶子节点所经过的各段路径所对应的 0 或者 1 连接起来就构成了字符的哈夫曼编码。

因为哈夫曼树具有不唯一性，因此哈夫曼编码也是不唯一的。构建哈夫曼树时，有些资料上会建议左孩子节点的权值应该不大于右孩子节点的权值，或者要求左孩子与右孩子节点权值大小关系应该保持一致。换句话说，就是要么保证所有左孩子节点权值小于或等于右孩子节点权值，要么保证所有右孩子节点权值小于或者等于左孩子节点权值。但我认为这并没有关系，也不需要保持左右孩子节点权值大小关系的一致性。

哈夫曼编码的实现代码可以直接放在前面介绍的 HFMTree 类中实现，增加 public 修饰的成员函数即可。

shikey.com转载分享

 复制代码

```
1 //生成哈夫曼编码
2 bool CreateHFMCode(int idx) //参数idx是用于保存哈夫曼树的数组某个节点的下标
3 {
4     //调用这个函数时，m_length应该已经等于整个哈夫曼树的节点数量，那么哈夫曼树的叶子节点数量应该
5     int leafNodeCount = (m_length + 1) / 2;
6
7     if (idx < 0 || idx >= leafNodeCount)
```




```

8  {
9      //只允许对叶子节点求其哈夫曼编码
10     return false;
11 }
12 string result=""; //保存最终生成的哈夫曼编码
13 int curridx = idx;
14 int tmpparent = m_data[curridx].parent;
15 while (tmpparent != -1) //沿着叶子向上回溯
16 {
17     if (m_data[tmpparent].lchild == curridx)
18     {
19         //前插0
20         result = "0" + result;
21     }
22     else
23     {
24         //前插1
25         result = "1" + result;
26     }
27     curridx = tmpparent;
28     tmpparent = m_data[curridx].parent;
29 } //end while
30 cout <<"下标为【"<< idx <<"】，权值为"<< m_data[idx].weight <<"的节点的哈夫曼编码是"<
31 return true;
32 }

```

在 main 主函数中，修改一下权值列表，完整的 main 主函数代码就是下面的样子。

 复制代码

```

1  int weighLst[] = { 12,15,9,24,8,32 };
2  HFMTTree hfmtobj(
3      sizeof(weighLst) / sizeof(weighLst[0]),    //权值列表中元素个数
4      weighLst                                     //权值列表首地址
5  );
6
7  hfmtobj.CreateHFMTTree(); //创建哈夫曼树
8  hfmtobj.preOrder(hfmtobj.GetLength()-1); //遍历哈夫曼树，参数其实就是根节点的下标（数组最
9
10 //求哈夫曼编码
11 cout <<"-----" << endl;
12 for(int i = 0; i < sizeof(weighLst) / sizeof(weighLst[0]); ++i)
13     hfmtobj.CreateHFMTCode(i);

```

执行结果如下：

```
100 41 17 8 9 24 59 27 12 15 32 -----
下标为【0】，权值为12的节点的哈夫曼编码是100
下标为【1】，权值为15的节点的哈夫曼编码是101
下标为【2】，权值为9的节点的哈夫曼编码是001
下标为【3】，权值为24的节点的哈夫曼编码是01
下标为【4】，权值为8的节点的哈夫曼编码是000
下标为【5】，权值为32的节点的哈夫曼编码是11
```

请注意，该结果与图 13 所展示的哈夫曼树以及图 15 所展示的哈夫曼编码结果不完全一样，这是因为程序编码中哈夫曼树的构建规则完全遵照“哈夫曼树左孩子节点的权值不大于右孩子节点的权值”，而 13 的哈夫曼树并没有遵照这个规则构建，例如节点 24 和 17 结合的时候，还有节点 32 和 27 结合的时候。

小结

这节课，我们先铺垫了几个概念，包括节点的权、从根节点到某节点的路径长度、节点的带权路径长度、树的带权路径长度，注意，这里要懂得怎么计算。由此，我们得出了哈夫曼树的概念：在含有 n 个带权叶节点的二叉树中，其中带权路径长度（WPL）最小的二叉树，也叫做**最优二叉树**。想要根据给定的 n 个叶子节点来构造哈夫曼树，要经历 4 个步骤：

将这 n 个节点分别作为 n 棵仅含一个节点（根节点）的二叉树，这样就构成了一个二叉树集合 F 。

构造一个新节点，在集合 F 中选取两棵根节点权值最小的树作为这个新节点的左右子树（谁左谁右顺序任意）构造一棵新二叉树。这个新节点的权值是左右两棵子树根节点的权值之和。

shikey.com转载分享

从集合 F 中删除刚刚选取的两棵树，并将新得到的树加入到集合 F 中。

重复上面的 2 个步骤，直到集合 F 中只剩下一棵树为止，这棵树就是哈夫曼树。

理清了思路，也就能理解我们这节课用数组来保存哈夫曼树的详细代码案例了。

在学习了哈夫曼树之后，就该了解哈夫曼树有什么用途了。哈夫曼树是用来创建哈夫曼编码的。哈夫曼编码是一种可以用于数据压缩的编码方式，哈夫曼编码的构造过程需要用到哈夫曼

树。

最后，我们详细讲解了对一段通信双方要传输的数据通过哈夫曼树进行哈夫曼编码的过程，同时给出了实现代码，希望通过这些代码能够帮你更深刻地理解哈夫曼树和哈夫曼编码的实现细节。

归纳思考

英文的 26 个字母使用的频率是不一样的，这 26 个字母的使用频率数据可以通过搜索引擎来搜索。如果要对这 26 个字母进行哈夫曼编码，计算一下使用哈夫曼编码可以对数据压缩多少。

欢迎你在留言区和我互动交流，如果觉得有所收获，也可以把课程分享给更多的朋友一起学习进步。我们下节课见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

shikey.com转载分享