哈夫曼树

原理　哈夫曼编码

通常的编码方法有固定长度编码和不等长编码两种。这是一个设计最优编码方案的问题，目的是使总码长度最短。这个问题是利用字符的使用频率来编码，是不等长编码方法，使得经常使用的字符编码较短，不常使用的字符编码较长。如果采用等长的编码方案，假设所有字符的编码都等长，则表示 n 个不同的字符需要⌈logn⌉位。例如 3 个不同的字符 a、b、c，至少需要两位二进制数表示，即 a:00、b:01、c:10。如果每个字符的使用频率都相等，则固定长度编码是空间效率最高的方法。

不等长编码方法需要解决两个关键问题：①编码尽可能短，我们可以让使用频率高的字符编码较短，使用频率低的字符编码较长，这种方法可以提高压缩率，节省空间，也能提高运算和通信速度，即频率越高，编码越短；②不能有二义性。

例如：ABCD 四个字符如果这样编码：

```
A: 0    B: 1    C: 01   D: 10
```

那么现在有一列数 0110，该怎样翻译呢？是翻译为 ABBA、ABD、CBA 还是 CD？这种混乱的译码如果用在军事情报中后果会很严重！

那么如何消除二义性呢？

解决的办法是：任何一个字符的编码都不能是另一个字符编码的前缀，即前缀码特性。

1952 年，数学家 D.A.Huffman 提出了一种最佳编码方式，被称为哈夫曼（Huffman）编码。哈夫曼编码很好地解决了上述两个关键问题，被广泛地应用于数据压缩，尤其是远距离通信和大容量数据存储。常用的 JPEG 图片就是采用哈夫曼编码压缩的。

哈夫曼编码的基本思想是以字符的使用频率作为权值构建一棵哈夫曼树，然后利用哈夫曼树对字符进行编码。构造一棵哈夫曼树，是将所要编码的字符作为叶子节点，将该字符在文件中的使用频率作为叶子节点的权值，以自底向上的方式，通过 n-1 次的“合并”运算后构造出的树。其核心思想是让权值大的叶子离根最近。

哈夫曼算法采取的**贪心策略**是，每次都从树的集合中取出没有双亲且权值最小的两棵树作为左、右子树，构造一棵新树，新树根节点的权值为其左、右孩子节点权值之和，将新树插入树的集合中。

1. 算法步骤

（1）确定合适的数据结构。编写程序前需要考虑的情况如下。

- 在哈夫曼树中，如果没有度为 1 的节点，则一棵有 n 个叶子节点的哈夫曼树共有 2n-1 个节点（n-1 次的“合并”，每次都产生一个新节点）。
- 构成哈夫曼树后，编码需要从叶子节点出发走一条从叶子到根的路径。译码需要从根出发走一条从根到叶子的路径。那么对于每个节点而言，需要知道每个节点的权值、双亲、左孩子、右孩子和节点的信息。

（2）初始化。构造 n 棵节点为 n 个字符的单节点树集合 $T=\{t_1,t_2,t_3,\ldots,t_n\}$，每棵树只有一个带权的根节点，权值为该字符的使用频率。

（3）如果在 T 中只剩下一棵树，则哈夫曼树构造成功，跳到第 6 步。否则，从集合 T 中取出没有双亲且权值最小的两棵树 t_i 和 t_j，将它们合并成一棵新树 z_k，新树的左孩子为 t_i，右孩子为 t_j，z_k 的权值为 t_i 和 t_j 的权值之和。

（4）从集合 T 中删去 t_i、t_j，加入 z_k。

（5）重复第（3）~（4）步。

（6）约定左分支上的编码为“0”，右分支上的编码为“1”。从叶子节点到根节点逆向求出每个字符的哈夫曼编码。那么从根节点到叶子节点路径上的字符组成的字符串为该叶子节点的哈夫曼编码，算法结束。

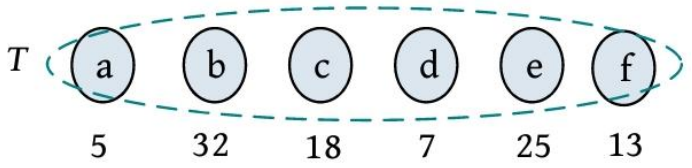
2.图解

假设一些字符及它们的使用频率如下表所示，那么如何得到它们的哈夫曼编码呢？

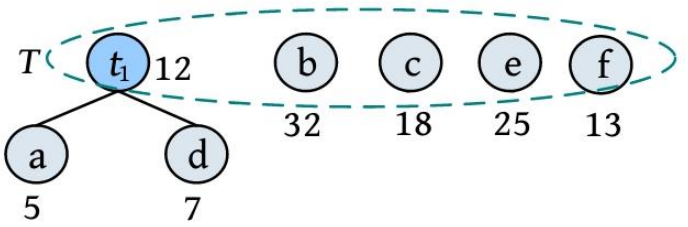
字符	a	b	c	d	e	f
频率	0.05	0.32	0.18	0.07	0.25	0.13

可以把每一个字符都作为叶子，将它们对应的频率作为其权值，因为只是比较大小，所以为了比较方便，可以对其同时扩大一百倍，得到 a:5、b:32、c:18、d:7、e:25、f:13。

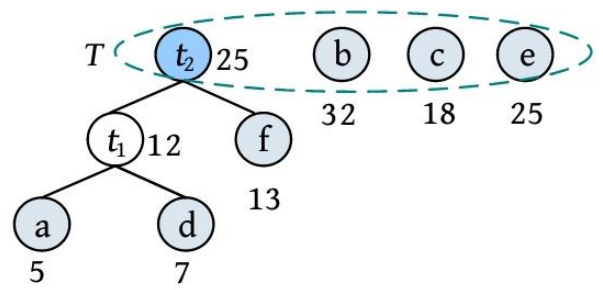
（1）初始化。构造 n 棵节点为 n 个字符的单节点树集合 T={a,b,c,d,e,f}，如下图所示。



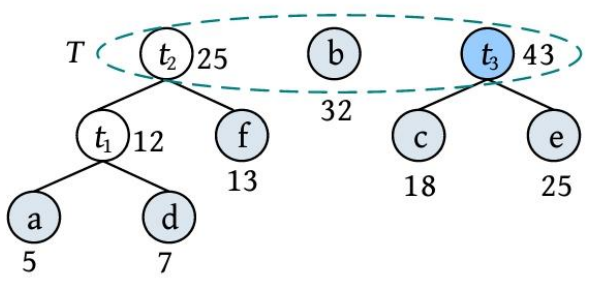
（2）从集合 T 中取出没有双亲且权值最小的两棵树 a 和 d，将它们合并成一棵新树 t_1，新树的左孩子为 a，右孩子为 d，新树的权值为 a 和 d 的权值之和 12。将新树的树根 t_1 加入集合 T，将 a、d 从集合 T 中删除，如下图所示。



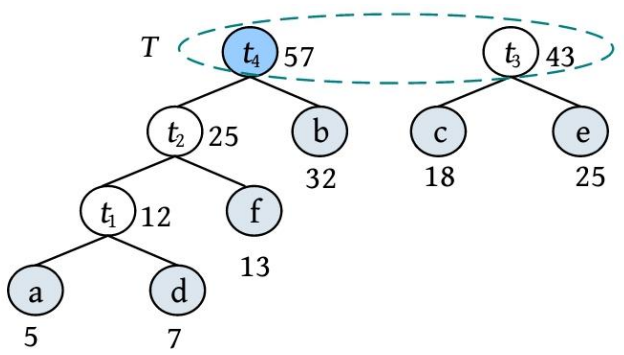
（3）从集合 T 中取出没有双亲且权值最小的两棵树 t_1 和 f，将它们合并成一棵新树 t_2，新树的左孩子为 t_1，右孩子为 f，新树的权值为 t_1 和 f 的权值之和 25。将新树的树根 t_2 加入集合 T，将 t_1 和 f 从集合 T 中删除，如下图所示。



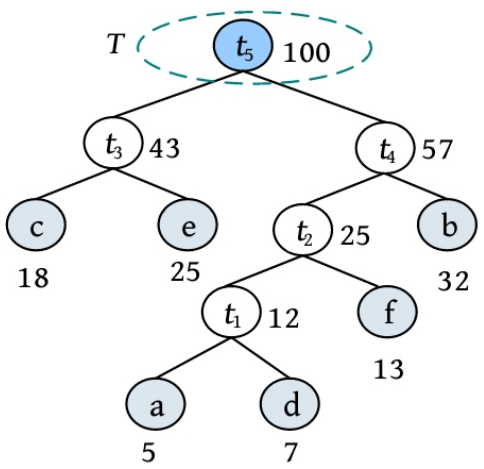
（4）从集合 T 中取出没有双亲且权值最小的两棵树 c 和 e，将它们合并成一棵新树 t_3，新树的左孩子为 c，右孩子为 e，新树的权值为 c 和 e 的权值之和 43。将新树的树根 t_3 加入集合 T，将 c 和 e 从集合 T 中删除，如下图所示。



（5）从集合 T 中取出没有双亲且权值最小的两棵树 t_2 和 b，将它们合并成一棵新树 t_4，新树的左孩子为 t_2，右孩子为 b，新树的权值为 t_2 和 b 的权值之和 57。新树的树根 t_4 加入集合 T，将 t_2 和 b 从集合 T 中删除，如下图所示。

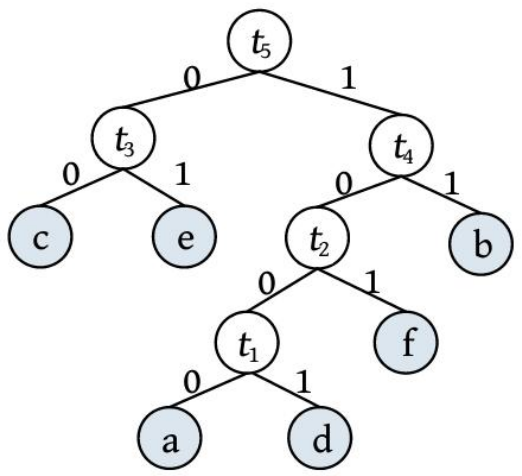


（6）从集合 T 中取出没有双亲且权值最小的两棵树 t_3 和 t_4，将它们合并成一棵新树 t_5，新树的左孩子为 t_3，右孩子为 t_4，新树的权值为 t_3 和 t_4 的权值之和 100。将新树的树根 t_5 加入集合 T，将 t_3 和 t_4 从集合 T 中删除，如下图所示。



（7）在集合 T 中只剩下一棵树，哈夫曼树构造成功。

（8）约定左分支上的编码为“0”，右分支上的编码为“1”。从叶子节点到根节点逆向求出每个字符的哈夫曼编码。那么从根节点到叶子节点路径上的字符组成的字符串为该叶子节点的哈夫曼编码，如下图所示。



a:1000　b:11　c:00　d:1001　e:01　f:101

3. 算法实现

在构造哈夫曼树的过程中，首先将每个节点的双亲、左孩子、右孩子都初始化为-1，找出所有节点中双亲为-1 且权值最小的两个节点 t_1、t_2，并合并为一棵二叉树，更新信息（双亲节点的权值为 t_1、t_2 权值之和，其左孩子为权值最小的节点 t_1，右孩子为次小的节点 t_2，t_1、t_2 的双亲为双亲节点的编号）。重复此过程，建成一棵哈夫曼树。

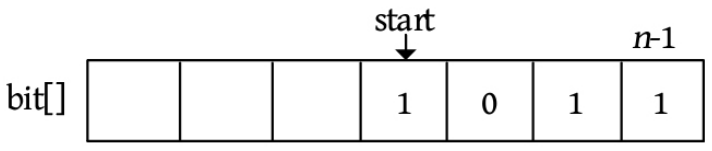
（1）数据结构。每个节点的结构都包括权值、双亲、左孩子、右孩子、节点字符信息五个域，如下图所示。

将其定义为结构体形式，定义节点结构体 HnodeType。

weight	parent	lchild	rchild	value

```
typedef struct{
   double weight; //权值
   int parent;  //双亲
   int lchild;  //左孩子
   int rchild;  //右孩子
   char value; //该节点表示的字符
} HNodeType;
```

在结构体的编码过程中，bit[]存放节点的编码，start 记录编码开始时的下标，在逆向编码（从叶子到根，想一想为什么不从根到叶子呢？）存储时，start 从 n-1 开始依次递减，从后向前存储；当读取时，从 start+1 开始到 n-1，从前向后输出，即该字符的编码，如下图所示。



编码结构体 HcodeType 代码如下。

```
typedef struct{
   int bit[MAXBIT]; //存储编码的数组
   int start; //编码开始下标
} HCodeType;
```

（2）初始化。初始化哈夫曼树数组 HuffNode[]中的节点权值为 0，双亲和左、右孩子均为-1，然后读入叶子节点的权值，如下表所示。

	weight	parent	lchild	rchild	value
0	5	−1	−1	−1	a
1	32	−1	−1	−1	b
2	18	−1	−1	−1	c
3	7	−1	−1	−1	d
4	25	−1	−1	−1	e
5	13	−1	−1	−1	f
6	0	−1	−1	−1	
7	0	−1	−1	−1	
8	0	−1	−1	−1	
9	0	−1	−1	−1	
10	0	−1	−1	−1	

（3）循环构造哈夫曼树。从集合 T 中取出双亲为-1 且权值最小的两棵树 t_i 和 t_j，将它们合并成一棵新树 z_k，新树的左孩子为 t_i，右孩子为 t_j，z_k 的权值为 t_i 和 t_j 的权值之和。

```
int i,j,x1,x2; //x1、x2 为两个最小权值节点的序号
double m1,m2; //m1、m2 为两个最小权值节点的权值
for(i=0;i<n-1;i++){
      m1=m2=MAXVALUE;  //初始化为最大值
      x1=x2=-1;  //初始化为-1
      //找出所有节点中权值最小、无双亲节点的两个节点
      for(j=0;j<n+i;j++){
         if(HuffNode[j].weight<m1 && HuffNode[j].parent==-1){
            m2=m1;
            x2=x1;
            m1=HuffNode[j].weight;
            x1=j;
         }
         else if(HuffNode[j].weight<m2 && HuffNode[j].parent==-1){
            m2=HuffNode[j].weight;
            x2=j;
         }
      }
      /* 更新新树信息 */
      HuffNode[x1].parent=n+i; //x1 的父亲为新节点编号 n+i
      HuffNode[x2].parent=n+i; //x2 的父亲为新节点编号 n+i
      HuffNode[n+i].weight=m1+m2; //新节点权值为两个最小权值之和 m1+m2
      HuffNode[n+i].lchild=x1; //新节点 n+i 的左孩子为 x1
      HuffNode[n+i].rchild=x2; //新节点 n+i 的右孩子为 x2
   }
}
```

图解：

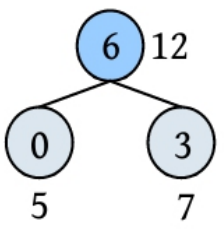
第 1 步，i=0 时：j=0; j<6；找双亲为-1 且权值最小的两个数。

```
x1=0   x2=3; //x1、x2 为两个最小权值节点的序号
m1=5  m2=7; //m1、m2 为两个最小权值节点的权值
HuffNode[0].parent=6;   //x1 的父亲为新节点编号 n+i
HuffNode[3].parent=6;   //x2 的父亲为新节点编号 n+i
HuffNode[6].weight=12;  //新节点权值为两个最小权值之和 m1+m2
HuffNode[6].lchild=0;   //新节点 n+i 的左孩子为 x1
HuffNode[6].rchild=3;   //新节点 n+i 的右孩子为 x2
```

数据更新后如下表所示：

	weight	parent	lchild	rchild	value
0	5	6	−1	−1	a
1	32	−1	−1	−1	b
2	18	−1	−1	−1	c
3	7	6	−1	−1	d
4	25	−1	−1	−1	e
5	13	−1	−1	−1	f
6	12	−1	0	3	
7	0	−1	−1	−1	
8	0	−1	−1	−1	
9	0	−1	−1	−1	
10	0	−1	−1	−1	

对应的哈夫曼树如下图所示：



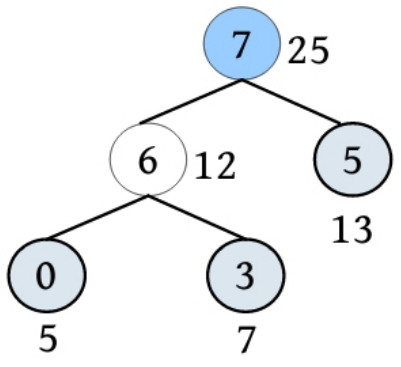
第 2 步，i=1 时：j=0; j<7；找双亲为-1 且权值最小的两个数。

```
x1=6   x2=5; //x1、x2 为两个最小权值节点的序号
m1=12  m2=13; //m1、m2 为两个最小权值节点的权值
HuffNode[5].parent=7;   //x1 的父亲为新节点编号 n+i
HuffNode[6].parent=7;   //x2 的父亲为新节点编号 n+i
HuffNode[7].weight=25;  //新节点权值为两个最小权值之和 m1+m2
HuffNode[7].lchild=6;   //新节点 n+i 的左孩子为 x1
HuffNode[7].rchild=5;   //新节点 n+i 的右孩子为 x2
```

数据更新后如下表所示：

	weight	parent	lchild	rchild	value
0	5	6	−1	−1	a
1	32	−1	−1	−1	b
2	18	−1	−1	−1	c
3	7	6	−1	−1	d
4	25	−1	−1	−1	e
5	13	7	−1	−1	f
6	12	7	0	3	
7	25	−1	6	5	
8	0	−1	−1	−1	
9	0	−1	−1	−1	
10	0	−1	−1	−1	

对应的哈夫曼树如下图所示：



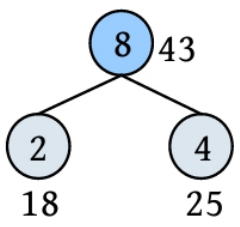
第 3 步，i=2 时：j=0; j<8；找双亲为-1 且权值最小的两个数。

```
x1=2    x2=4; //x1、x2 为两个最小权值节点的序号
m1=18  m2=25; //m1、m2 为两个最小权值节点的权值
HuffNode[2].parent=8;   //x1 的父亲为新节点编号 n+i
HuffNode[4].parent=8;   //x2 的父亲为新节点编号 n+i
HuffNode[8].weight=43;  //新节点权值为两个最小权值之和 m1+m2
HuffNode[8].lchild=2;   //新节点 n+i 的左孩子为 x1
HuffNode[8].rchild=4;   //新节点 n+i 的右孩子为 x2
```

数据更新后如下表所示：

	weight	parent	lchild	rchild	value
0	5	6	−1	−1	a
1	32	−1	−1	−1	b
2	18	8	−1	−1	c
3	7	6	−1	−1	d
4	25	8	−1	−1	e
5	13	7	−1	−1	f
6	12	7	0	3	
7	25	−1	6	5	
8	43	−1	2	4	
9	0	−1	−1	−1	
10	0	−1	−1	−1	

对应的哈夫曼树如下图所示：



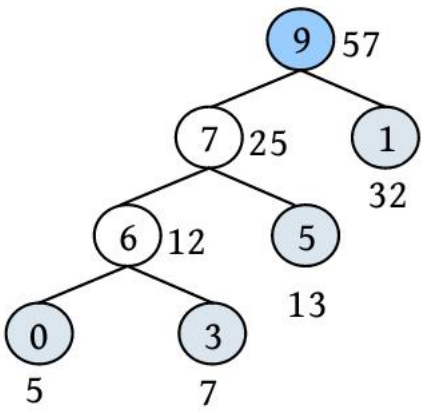
第 4 步，i=3 时：j=0; j<9；找双亲为-1 且权值最小的两个数。

```
x1=7    x2=1; //x1、x2 为两个最小权值节点的序号
m1=25  m2=32; //m1、m2 为两个最小权值节点的权值
HuffNode[7].parent=9;   //x1 的父亲为新节点编号 n+i
HuffNode[1].parent=9;   //x2 的父亲为新节点编号 n+i
HuffNode[9].weight=57;  //新节点权值为两个最小权值之和 m1+m2
HuffNode[9].lchild=7;   //新节点 n+i 的左孩子为 x1
HuffNode[9].rchild=1;   //新节点 n+i 的右孩子为 x2
```

数据更新后如下表所示：

	weight	parent	lchild	rchild	value
0	5	6	−1	−1	a
1	32	9	−1	−1	b
2	18	8	−1	−1	c
3	7	6	−1	−1	d
4	25	8	−1	−1	e
5	13	7	−1	−1	f
6	12	7	0	3	
7	25	9	6	5	
8	43	−1	2	4	
9	57	−1	7	1	
10	0	−1	−1	−1	

对应的哈夫曼树如下图所示：



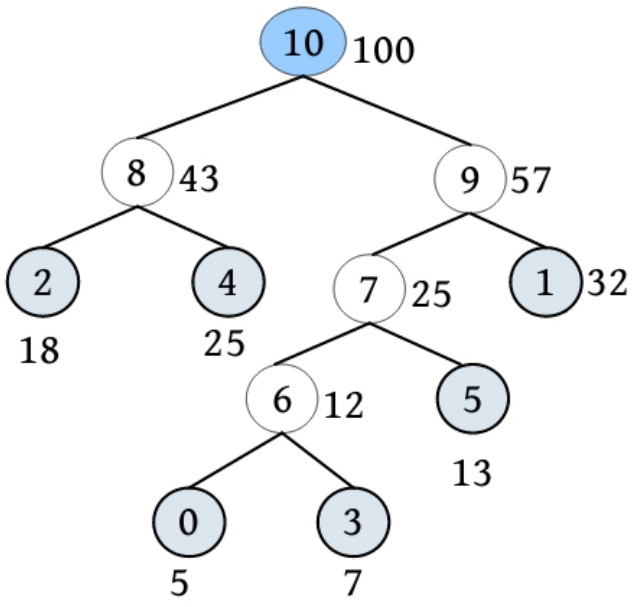
第 5 步，i=4 时：j=0; j<10；找双亲为-1 且权值最小的两个数：

```
x1=8    x2=9; //x1、x2 为两个最小权值节点的序号
m1=43  m2=57; //m1、m2 为两个最小权值节点的权值
HuffNode[8].parent=10;    //x1 的父亲为生成的新节点编号 n+i
HuffNode[9].parent=10;    //x2 的父亲为生成的新节点编号 n+i
HuffNode[10].weight=100;  //新节点权值为两个最小权值之和 m1+m2
HuffNode[10].lchild=8;    //新节点编号 n+i 的左孩子为 x1
HuffNode[10].rchild=9;    //新节点编号 n+i 的右孩子为 x2
```

数据更新后如下表所示：

	weight	parent	lchild	rchild	value
0	5	6	−1	−1	a
1	32	9	−1	−1	b
2	18	8	−1	−1	c
3	7	6	−1	−1	d
4	25	8	−1	−1	e
5	13	7	−1	−1	f
6	12	7	0	3	
7	25	9	6	5	
8	43	10	2	4	
9	57	10	7	1	
10	100	−1	8	9	

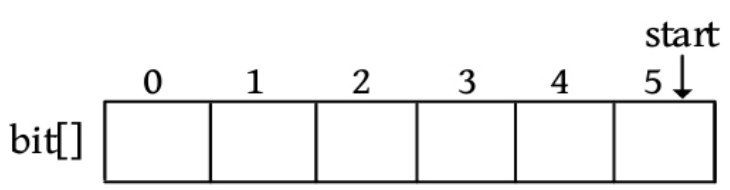
对应的哈夫曼树如下图所示：



（4）输出哈夫曼编码。

```
void HuffmanCode(HCodeType HuffCode[MAXLEAF], int n){
   HCodeType cd;      /* 定义一个临时变量来存放求解编码时的信息 */
   int i,j,c,p;
   for(i=0;i<n;i++){
      cd.start=n-1;
      c=i;  //i 为叶子节点编号
      p=HuffNode[c].parent;
      while(p!=-1){
         if(HuffNode[p].lchild==c){
            cd.bit[cd.start]=0;
         }
         else
            cd.bit[cd.start]=1;
         cd.start--;       /* start 向前移动一位 */
         c=p;           /* c、p 变量上移，准备下一循环 */
         p=HuffNode[c].parent;
      }
  /* 把叶子节点的编码信息从临时编码 cd 中复制出来，放入编码结构体数组 */
      for(j=cd.start+1;j<n;j++)
         HuffCode[i].bit[j]=cd.bit[j];
      HuffCode[i].start=cd.start;
   }
}
```

哈夫曼编码数组如下图所示：



第 1 步，i=0 时：c=0。

构造好的哈夫曼树数组如下表所示：

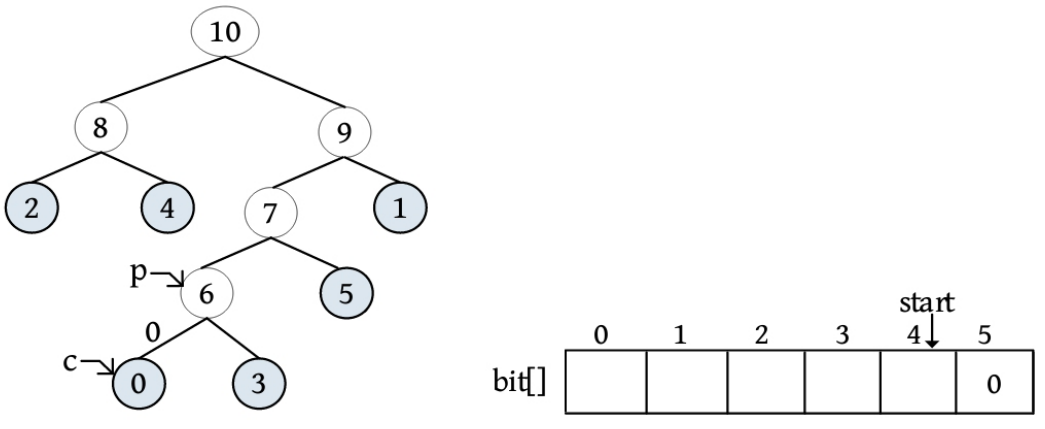
	weight	parent	lchild	rchild	value
0	5	6	−1	−1	a
1	32	9	−1	−1	b
2	18	8	−1	−1	c
3	7	6	−1	−1	d
4	25	8	−1	−1	e
5	13	7	−1	−1	f
6	12	7	0	3	
7	25	9	6	5	
8	43	10	2	4	
9	57	10	7	1	
10	100	−1	8	9	

如果 p!=-1，那么从表 HuffNode[]中读出节点 6 的左孩子和右孩子，判断节点 0 是它的左孩子还是右孩子；如果是左孩子，则编码为 0；如果是右孩子，则编码为 1。

从上表中可以看出：

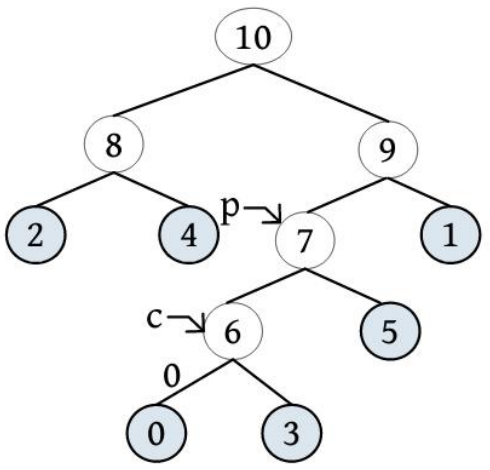
```
HuffNode[6].lchild=0;// 节点 0 是其父亲节点 6 的左孩子
cd.bit[5]=0;//编码为 0
cd.start--=4; /* start 向前移动一位*/
```

哈夫曼树和哈夫曼编码数组如下图所示：



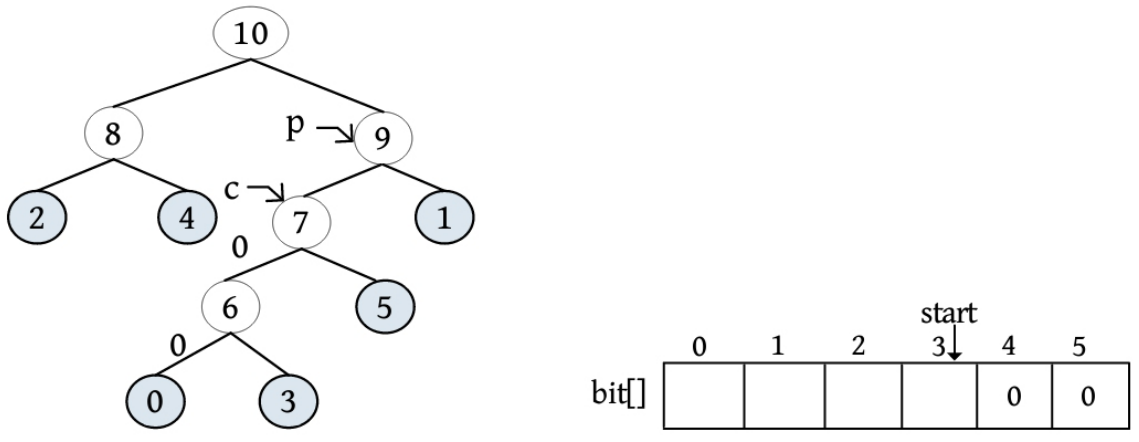

```
c=p=6;              /* c、p 变量上移，准备下一循环 */
p=HuffNode[6].parent=7;
```

c、p 变量上移后如下图所示：



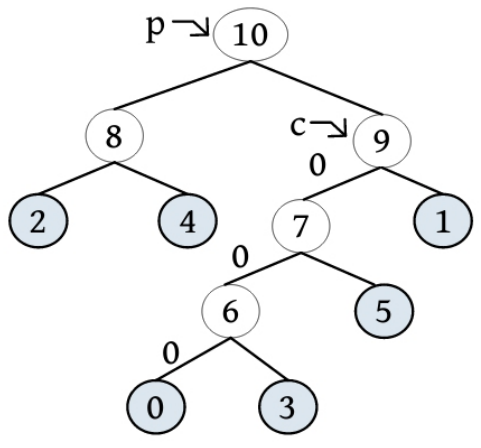

```
p!=-1;
HuffNode[7].lchild=6;// 节点 6 是其父亲节点 7 的左孩子
cd.bit[4]=0;//编码为 0
cd.start--=3;       /* start 向前移动一位*/
c=p=7;          /* c、p 变量上移，准备下一循环 */
p=HuffNode[7].parent=9;
```

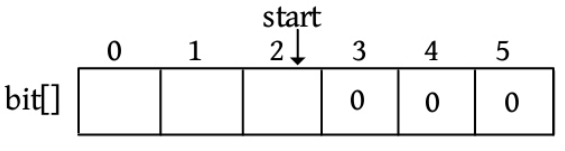
哈夫曼树和哈夫曼编码数组如下图所示：




```
p!=-1;
HuffNode[9].lchild=7;// 节点 7 是其父亲节点 9 的左孩子
cd.bit[3]=0;//编码为 0
cd.start--=2;       /* start 向前移动一位*/
c=p=9;          /* c、p 变量上移，准备下一循环 */
p=HuffNode[9].parent=10;
```

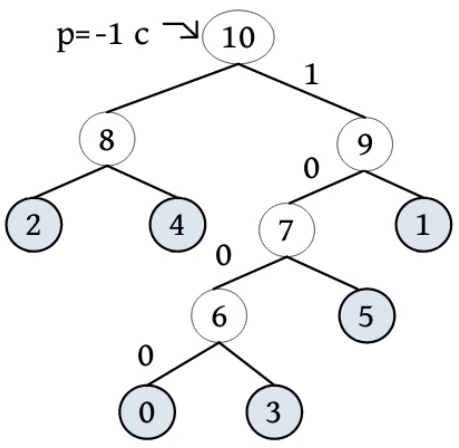
哈夫曼树和哈夫曼编码数组如下图所示：

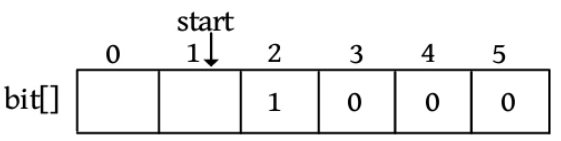





```
p!=-1;
HuffNode[10].lchild!=9;// 节点 9 不是其父亲节点 10 的左孩子
cd.bit[2]=1;//编码为 1
cd.start--=1;       /* start 向前移动一位*/
c=p=10;           /* c、p 变量上移，准备下一循环 */
p=HuffNode[10].parent=-1;
```

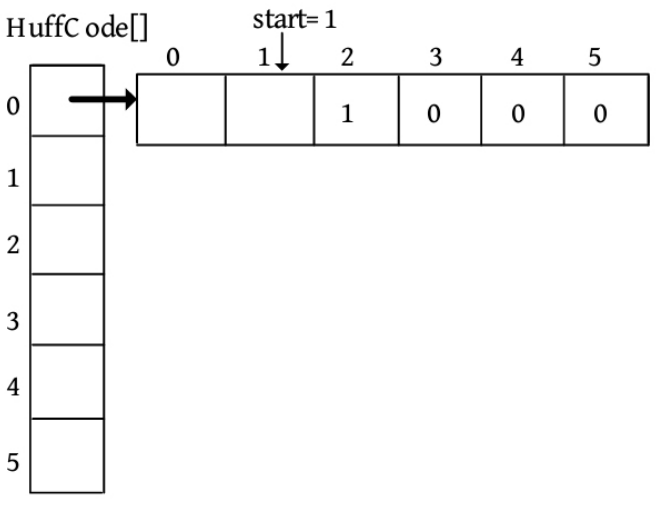
哈夫曼树和哈夫曼编码数组如下图所示：






```
p=-1;该叶子节点编码结束
/* 把叶子节点的编码信息从临时编码 cd 中复制出来，放入编码结构体数组 */
for(j=cd.start+1; j<n; j++)
   HuffCode[i].bit[j]=cd.bit[j];
HuffCode[i].start=cd.start;
```

HuffCode[]数组如下图所示（图中的箭头不表示指针）：



4. 算法分析

时间复杂度：由程序可以看出，在函数 HuffmanTree()中，“if(HuffNode[j].weight<m1&& HuffNode[j].parent==−1)”为基本语句，外层 i 与 j 组成双层循环。

- i=0 时，该语句执行 n 次。
- i=1 时，该语句执行 n+1 次。
- i=2 时，该语句执行 n+2 次。
- ……

- i=n−2 时，该语句执行 n+(n−2)次。

基本语句共执行 n+(n+1)+(n+2)+…+(n+(n−2))=(n−1)×(3n−2)/2 次。在函数 HuffmanCode()中，编码和输出编码的时间复杂度都接近 n^2，则该算法时间复杂度为 O(n^2)。

空间复杂度：所需存储空间为节点结构体数组与编码结构体数组，哈夫曼树数组 HuffNode[]中的节点为 n 个，每个节点都包含 bit[MAXBIT]和 start 两个域，则该算法的空间复杂度为 O(n×MAXBIT)。

5. 算法优化

该算法可以从以下两个方面进行优化。

（1）在函数 HuffmanTree()中找两个权值最小节点时，使用优先队列，时间复杂度为 logn，执行 n−1 次，总时间复杂度为 O(nlogn)。

（2）在函数 HuffmanCode()中，在哈夫曼编码数组 HuffNode[]中可以定义一个动态分配空间的线性表来存储编码，每个线性表的长度都为实际的编码长度，这样可以大大节省空间。