目录

[基本介绍 2](#_Toc438651692)

[例子与演算过程 2](#_Toc438651693)

[（一）进行霍夫曼编码前，我们先创建一个霍夫曼树。 2](#_Toc438651694)

[（二）进行编码 3](#_Toc438651695)

[数据压缩 4](#_Toc438651696)

[数据解压缩 5](#_Toc438651697)

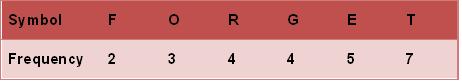
[代码 5](#_Toc438651698)

# 基本介绍

哈夫曼编码是一种编码方式，是一种用于无损数据压缩的熵编码。哈夫曼编码使用变长的编码表对源符号进行编码，其中变长编码表是通过一种评估来源符号出现概率的方法得到的。霍夫曼树常处理符号编写工作。根据整组数据中符号出现的频率高低，决定如何给符号编码。如果符号出现的频率太高，则给符号的码越短，相反符号的号码越长。

# 例子与演算过程

假设我们要给一个英文单字"F O R G E T"进行霍夫曼编码



Figure

### （一）进行霍夫曼编码前，我们先创建一个霍夫曼树。

⒈将每个英文字母依照出现频率由小排到大，最小在左，如Figure 1。

⒉每个字母都代表一个终端节点（叶节点），比较F.O.R.G.E.T五个字母中每个字母的出现频率，将最小的两个字母频率相加合成一个新的节点。发现F与O的频率最小，故相加2+3=5。

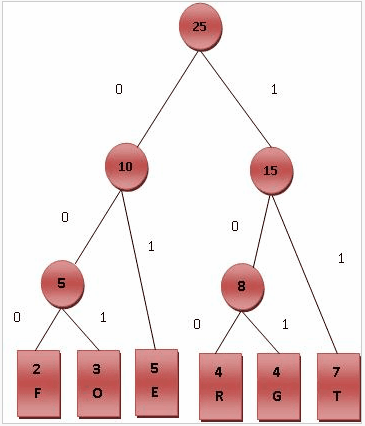
⒊比较5.R.G.E.T，发现R与G的频率最小，故相加4+4=8。

⒋比较5.8.E.T，发现5与E的频率最小，故相加5+5=10。

⒌比较8.10.T，发现8与T的频率最小，故相加8+7=15。

⒍最后剩10.15，没有可以比较的对象，相加10+15=25。

最后产生的树状图就是霍夫曼树。

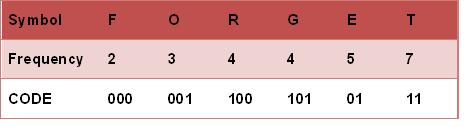


Figure

### （二）进行编码

1.给霍夫曼树的所有左链接'0'与右链接'1'。

2.从树根至树叶依序记录所有字母的编码，如Figure 3。



Figure

# 数据压缩

实现[霍夫曼编码](https://zh.wikipedia.org/wiki/%E9%9C%8D%E5%A4%AB%E6%9B%BC%E7%B7%A8%E7%A2%BC)的方式主要是创建一个[二叉树](https://zh.wikipedia.org/wiki/%E4%BA%8C%E5%85%83%E6%A8%B9)和其节点。这些树的节点可以存储在[数组](https://zh.wikipedia.org/wiki/%E9%99%A3%E5%88%97)里，[数组](https://zh.wikipedia.org/wiki/%E9%99%A3%E5%88%97)的大小为符号（symbols）数的大小n，而节点分为是终端节点（叶节点）与非终端节点（内部节点）。

一开始，所有的节点都是终端节点，节点内有三个字段：

1.符号（Symbol）

2.权重（Weight、Probabilities、Frequency）

3.指向父节点的[链接](https://zh.wikipedia.org/wiki/%E9%80%A3%E7%B5%90%E4%B8%B2%E5%88%97)（Link to its parent node）

而非终端节点内有四个字段：

1.权重（Weight、Probabilities、Frequency）

2.指向两个子节点的 [链接](https://zh.wikipedia.org/wiki/%E9%80%A3%E7%B5%90%E4%B8%B2%E5%88%97)（Links to two child node）

3.指向父节点的[链接](https://zh.wikipedia.org/wiki/%E9%80%A3%E7%B5%90%E4%B8%B2%E5%88%97)（Link to its parent node）

基本上，我们用'0'与'1'分别代表指向左子节点与右子节点，最后为完成的二叉树共有n个终端节点与n-1个非终端节点，去除了不必要的符号并产生最佳的编码长度。

过程中，每个终端节点都包含着一个权重（Weight、Probabilities、Frequency），两两终端节点结合会产生一个新节点，新节点的权重是由两个权重最小的终端节点权重之总和，并持续进行此过程直到只剩下一个节点为止。

我使用两个队列（Queue）创件霍夫曼树。第一个队列用来存储n个符号（即n个终端节点）的权重，第二个队列用来存储两两权重的合（即非终端节点）。此法可保证第二个队列的前端（Front）权重永远都是最小值，且方法如下：

⒈把n个终端节点加入第一个队列（依照权重大小排列，最小在前端）

⒉如果队列内的节点数>1，则：

⑴从队列前端移除两个最低权重的节点

⑵将（1）中移除的两个节点权重相加合成一个新节点

⑶加入第二个队列

⒊最后在第一个队列的节点为根节点

虽然使用此方法比使用优先队列的时间复杂度还低，但是注意此法的第1项，节点必须依照权重大小加入队列中，如果节点加入顺序不按大小，则需要经过排序，则至少花了O（n log n）的时间复杂度计算。

但是在不同的状况考量下，时间复杂度并非是最重要的，如果我们今天考虑英文字母的出现频率，变数n就是英文字母的26个字母，则使用哪一种算法时间复杂度都不会影响很大，因为n不是一笔庞大的数字。

# 数据解压缩

简单来说，霍夫曼码树的解压缩就是将得到的前置码（Prefix Huffman code）转换回符号，通常借由树的追踪（Traversal），将接收到的比特串（Bits stream）一步一步还原。但是要追踪树之前，必须要先重建霍夫曼树 ；某些情况下，如果每个符号的权重可以被事先预测，那么霍夫曼树就可以预先重建，并且存储并重复使用，否则，发送端必须预先发送霍夫曼树的相关信息给接收端。

最简单的方式，就是预先统计各符号的权重并加入至压缩之比特串，但是此法的运算量花费相当大，并不适合实际的应用。若是使用Canonical encoding，则可精准得知树重建的数据量只占B2^B比特（其中B为每个符号的比特数（bits））。如果简单将接收到的比特串一个比特一个比特的重建，例如：'0'表示父节点，'1'表示终端节点，若每次读取到1时，下8个比特则会被解读是终端节点（假设数据为8-bit字母），则霍夫曼树则可被重建，以此方法，数据量的大小可能为2~320字节不等。虽然还有很多方法可以重建霍夫曼树，但因为压缩的数据串包含"traling bits"，所以还原时一定要考虑何时停止，不要还原到错误的值，如在数据压缩时时加上每笔数据的长度等。

但是由于技术问题没有真正的实现压缩的功能，知识将二进制的代码当作字符串来输出，以展示效果。

# 代码

#include <map>

#include <queue>

#include <vector>

#include <string>

#include <cstring>

#include <cstdio>

#include <iostream>

#include <cstdlib>

#include <algorithm>

#include <cmath>

using namespace std;

class Huffman

{

private:

map<char, int> mp;

map<char, string> rmp;

struct Node

{

char ch;

int cnt;

Node \*lson, \*rson;

bool isLeaf;

Node(){}

Node(char ch, int cnt, Node\* l, Node\* r, bool b):ch(ch),cnt(cnt),lson(l),rson(r),isLeaf(b){}

};

void dfs(Node \*now, string str)

{

if(now->isLeaf == true)

{

cout << now->ch << " : " << str << endl;

rmp[now->ch] = str;

}

if(now->lson != NULL)

{

dfs(now->lson, str + "0");

}

if(now->rson != NULL)

{

dfs(now->rson, str + "1");

}

}

public:

Node \*head, \*now;

void input(char ch)

{

if(mp.find(ch) == mp.end())

mp[ch] = 0;

mp[ch] ++;

}

static bool cmp(Node \*a, Node \*b){return a->cnt < b->cnt;}

void build()

{

deque<Node\*> que1,que2;

que1.clear(), que2.clear();

for(pair<char ,int> x : mp)

{

cout << x.first << endl;

Node\* tmp = new Node(x.first, x.second, NULL, NULL, true);

que1.push\_back(tmp);

}

sort(que1.begin(), que1.end(), cmp);

int len1, len2;

Node\* tmp[2];

len1 = que1.size(),len2 = que2.size();

while(len1 + len2 > 1)

{

for(int i = 0; i < 2; i++)

{

if(len1 == 0)

{

tmp[i] = que2[0];

que2.pop\_front();

len2--;

}

else if(len2 == 0)

{

tmp[i] = que1[0];

que1.pop\_front();

len1--;

}

else if(que1[0]->cnt < que2[0]->cnt)

{

tmp[i] = que1[0];

que1.pop\_front();

len1--;

}

else

{

tmp[i] = que2[0];

que2.pop\_front();

len2--;

}

}

Node\* node = new Node('\0', tmp[0]->cnt+tmp[1]->cnt, tmp[0], tmp[1], false);

que2.push\_back(node);

len2 ++;

}

head = que2[0];

}

map<char, string> getmap()

{

string str = "";

dfs(head, str);

return rmp;

}

char unzipin(char ch)

{

if(ch == '0')

{

now = now->lson;

if(now->isLeaf)

{

char ret = now->ch;

now = head;

return ret;

}

}

else

{

now = now->rson;

if(now->isLeaf)

{

char ret = now->ch;

now = head;

return ret;

}

}

return '\0';

}

}huffman;

int main()

{

cout << "!!!!" << '\0'<< "!!!!" << endl;

freopen("data.in", "r", stdin);

// string str = "FFOOORRRRGGGGEEEEETTTTTTT";

// for(int i = 0; i < str.length(); i++)

// huffman.input(str[i]);

char ch;

while(scanf("%c", &ch) != EOF)

huffman.input(ch);

fclose(stdin);

huffman.build();

map<char, string> mp = huffman.getmap();

freopen("data.in", "r", stdin);

freopen("code.out", "w", stdout);

while(scanf("%c", &ch)!=EOF)

{

cout << mp[ch];

}

fclose(stdin);

fclose(stdout);

freopen("code.out", "r", stdin);

freopen("unzip.out", "w", stdout);

huffman.now = huffman.head;

while(scanf("%c", &ch)!=EOF)

{

// printf("%c", huffman.unzipin(ch));

char tmp = huffman.unzipin(ch);

if(tmp != '\0') printf("%c", tmp);

}

fclose(stdout);

fclose(stdin);

// huffman.show();

return 0;

}