

# 北京邮电大学

## 实验报告



实验名称: 基于 Huffman 树实现的解压缩文件

班级 : 2019211309    姓名 : 陈悦    学号 : 2019211413    分工 : 文档

班级 : 2019211309    姓名 : 马晓亮    学号 : 2019211400    分工 : 代码

2020 年 11 月 29 日

## 一 需求分析

本次实验要求设计一个解压缩程序。该程序要实现两个功能，首先为压缩任意文件，压缩后的文件包括解压所需的信息和压缩后的文件本体两部分。解压功能则需要利用压缩文件所给的信息，将文件内容还原为原来的文件。

## 二 概要设计

本程序的函数调用关系如下

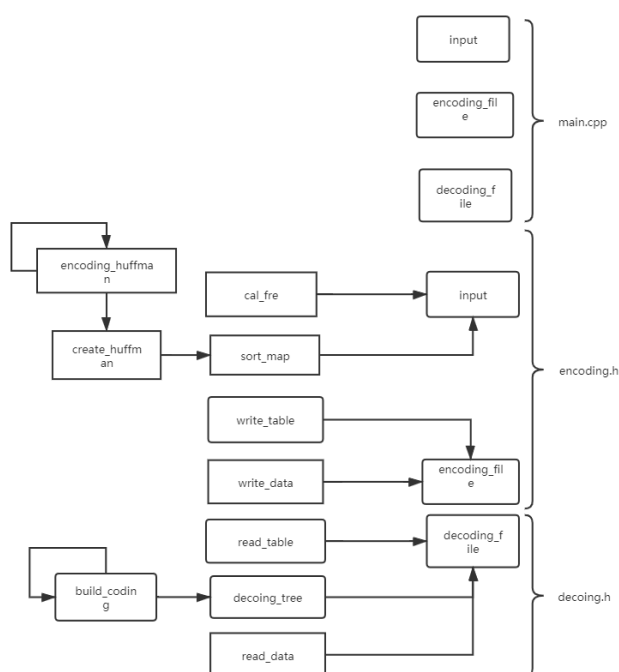


图 1: 函数调用关系图

该程序需要用到两个数据结构

首先是哈夫曼树

1. int flag 该节点是否为叶子结点
2. unsigned char val 该节点存有的值
3. int fre 该节点的频率
4. struct HuffmanTree \*ld,\*rd 指向左右子树的指针

再是解压表中元素的表头

- 1. unsigned char size 编码后对应字串的长度
- 2. unsigned char byte 原 8 位字串

在压缩后的文件中，文件的结构如下

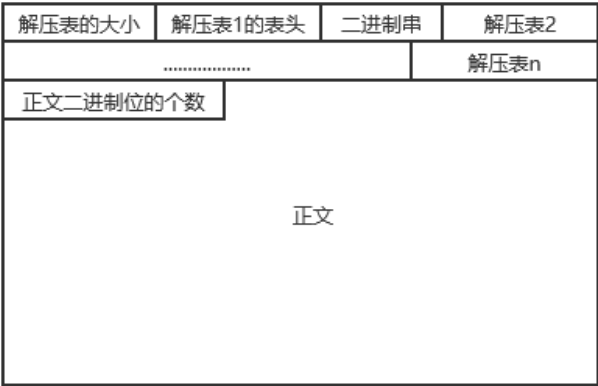


图 2: 文件结构图

每一个解压表由一个表头和一个二进制串构成，表头包含二进制串的长度与二进制串对应字符。

### 三 详细设计

---

**Algorithm 1** HuffmanTree 建树

---

```
1: 从解码表还原哈夫曼树
2: function BUILD_DECODING(node, i, s)
3:   if s 为空 then
4:     初始化 node 为值为 i 的叶子结点
5:   end if
6:   if s[0] == 0 then
7:     s 前进一位
8:     初始化 node->ld
9:     build_decodingnode->ld, i, s
10:  end if
11:  if s[0] == 1 then
12:    s 前进一位
13:    初始化 node->rd
14:    build_decodingnode->rd, i, s
15:  end if
16: end function
17: 利用优先队列建立哈夫曼树
18: function CREATE_HUFFMAN
19:   while 优先队列中还有一个以上的元素 do
20:     从优先队列中取出两个哈夫曼树
21:     将两个哈夫曼树合并后放回优先队列
22:   end while
23: end function
```

---

---

**Algorithm 2** 建立压缩表

---

```
1: 利用哈夫曼树建立压缩表
2: function ENCODING_TREE(rt, str)
3:   if rt 是一个叶子结点 then
4:     table[rt->val] = str
5:     二进制位的总数加上  $str.length() * rt->fre$ 
6:   end if
7:   if rt 的左子树不为空 then
8:     encoding_tree(rt->ld, str + 0)
9:   end if
10:  if rt 的右子树不为空 then
11:    encoding_tree(rt->rd, str + 1)
12:  end if
13: end function
```

---

---

**Algorithm 3** 压缩文件的流程

---

- 1: 调用 `input()` 正文信息, 调用 `calu_fre()` 统计频率
  - 2: 利用得到的频率初始化叶子结点, 调用 `sort_map()`, 将叶子结点压入优先队列中
  - 3: 调用 `create_huffman()` 建立哈夫曼树, 并且调用 `encoding_tree()` 得到压缩表
  - 4: 将压缩表以解压表的格式写入文件中
  - 5: 统计正文的字节数, 值放在 `size` 中
  - 6: 将 `size` 写入文件中
  - 7: **while**  $128 \leq size$  **do**
  - 8:     `size` 减去 128, 并读取 128 个字符存入到 `line` 中
  - 9:     调用 `write_data(line)`, 将 128 压缩后写入文件中
  - 10: **end while**
  - 11: 将剩下的字符单个读取, 并压缩写入文件中
- 

---

**Algorithm 4** 解压文件的流程

---

- 1: 调用 `read_table()` 读入解压表
  - 2: `decoding_tree()`, 将解压表转为哈夫曼树, 便于解压
  - 3: 读入正文二进制位的个数
  - 4: 调用 `read_data()`, 读入数据并解压写入到文件中
-

## 四 调试分析报告

在压缩算法中我们使用了优先队列算法建树，此处的算法复杂度为  $O(k \log k)$ ，然后我们再将正文压缩，复杂度为  $O(n)$  所以压缩算法的复杂度为  $O(k \log k + n)$ ，其中  $n$  为正文的长度， $k$  为正文中出现字符的种类，一般认为  $n$  远大于  $k$ ，则算法复杂度可以视为  $O(n)$

解压算法的复杂度为文件大小，即也为  $O(n)$

## 五 用户使用说明

用户可以使用 IDE 或者手动编译源代码 `stack.cpp`，获得可执行文件。

笔者使用的 gcc 版本为 8.1.0

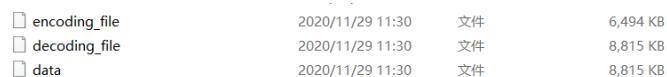
开始程序后，用户需要输入两个参数，0 表示压缩，1 表示解压。

在压缩模式下，程序会将当前文件夹下的 `data` 文件压缩到 `encoding_file` 中。

在解压模式下，程序会将当前文件夹下的 `encoding_file` 文件解压到 `decoding_file` 中。

## 六 测试结果

为了测试程序压缩的效果和正确性，我们从互联网上下载了《斗破苍穹》，文件大小为 8,815KB。经我们的程序压缩后，大小为 6,494KB，压缩为原来的  $1/4$ 。



encoding_file	2020/11/29 11:30	文件	6,494 KB
decoding_file	2020/11/29 11:30	文件	8,815 KB
data	2020/11/29 11:30	文件	8,815 KB

图 3: 压缩效果

然后再将压缩后的文件解压，得到的文件与原文件对比相同。