# Huffman压缩

PB20111686　黄瑞轩

## 实验要求

在合适的情况下，利用Huffman编码对文件进行压缩可以减少其占用空间，同时在需要使用到文件的时候也可以根据压缩文件中所提供的信息来将其还原为原文件。本次实验中，我们将实现一个基于Huffman编码的文件压缩/解压缩工具。

## 设计思路

本实验中采用经典的Huffman树构造模式，对文件以流的形式进行字符统计，并将统计结果和压缩后的流存放在压缩文件中。解压时软件不需要获得更多的信息，仅通过压缩文件就可实现对文件的解压。

本实验中我设计了`BitQueue`、`Heap`、`HuffmanNode`、`HuffmanTree`四个类用作依赖数据结构。出于性能优化设计，在统计字符频数以及字符编码时，使用了散列表。

`BitQueue`是一个布尔值队列，其作用是以布尔链表的形式储存和输出密码流，所含成员及定义如下。

```
class BitQueue : public deque<bool> {
public:
    void push_b_s(string s); // 将一个01 string类型字符串压入队列中
    string pop_b_s();        // 弹出一个字节，如果不足一个字节，返回空
    string pop_b_s(uint len);// 弹出位宽为len的一个字符串
    char pop_bit();          // 弹出一位
};
```

`Heap`是一个小根堆，其作用是根据`HuffmanNode`的权值来排序，所含成员及定义如下。

```
class Heap{
public:
    HuffmanNode** e_a = nullptr;              // 起始地址
    uint capacity = 0;                        // 载量
    uint ptr = 0;                             // 已用空间

    // 堆初始化函数
    Heap(uint capacity);
    // 交换两个节点a,b的所有信息
    void swap(HuffmanNode* a, HuffmanNode* b);
    // 向堆中插入一个节点
    bool insert(HuffmanNode* node);
    // 获取堆顶元素并从堆中删除
    HuffmanNode* pop();
    // 堆化
    void heapify();
};
```

`HuffmanNode`是一个用于储存信息的结构体，所含成员及定义如下。

```
class HuffmanNode{
public:
    uint w = 0;                     //权重
    uint name = 0;                      //表示的字符(uint序号)
    std::string code = "";              //其编码
    std::vector<HuffmanNode*> child;    //孩子链表
    uint fork_num = 2;                  //n叉树
    uint child_num = 0;                 //指示此节点孩子数量，同时也指示下一个插入的孩子的下标
    bool is_virtual = false;
```

```
    //非叶子节点初始化函数
    HuffmanNode(uint w, uint fork_num, bool is_virtual = false);
    //叶子节点初始化函数
    HuffmanNode(uint w, uint name, uint fork_num, bool is_virtual = false);
    //层序遍历
    void FloorTraverse(HuffmanNode* node);
    //插入孩子，返回是否成功插入
    bool InsertChild(HuffmanNode* node);
    //把当前节点的权重计算为所有孩子的权重之和
    void CaculateWeight();
    //将code所代表的n进制string转化为2进制string
    void Regularize();
};
```

HuffmanTree是Huffman树的数据结构，所含成员及定义如下。

```
class HuffmanTree{
public:
    uint fork_num = 2;              //叉数
    map<uint, string> code_map;     //first->字符代表的数字，second->字符被编的码
    HuffmanNode* root = nullptr;    //Huffman树根
    uint header_bytes = 8;          //文件头增加的字节数
    uint extra_bits = 0;            //文件尾部多加的0bit数
    uint byte_type = 8;             //读取的时候一个字是多少位

    HuffmanTree(uint Scan_bits, uint fork_num);
    //将此树全部编码，存于code[i]中
    void encode_helper(HuffmanNode* node);
    void insert_encoded_leaf_to_map(HuffmanNode* node);
    void encode(HuffmanNode* node);
    //根据堆h来建n叉树
    void build(Heap* h, uint n);
    //展示编码
    void display_encode();
    //统计存这个huffman树需要多少额外的字节
    void stat_extra_bytes();
    //将树保存成html代码以供展示
    void save_tree(string new_file_name);
};
```

压缩文件的设计：

- 文件头（2 byte）正常压缩：&&，这里&代表不可打印字符137；未被压缩：无文件头；
- 第三个配置字节：表示对于还原出来的文件，最后一个读写单位其实最后面补了多少个字节的0，值域：0~2，用2bit存；写入的整个文件的二进制流，最后因为不满一个字节而补了多少个bit的0，值域：0~7，用3bit存；存读取方式，值域：0~3，[0]：0.5字节；[1]：1字节；[2]：2字节；[3]：3字节，用2bit存；
- 接下来四个字节，表示此文件中出现了多少个Huffman编码，按无符号数解释；
- 然后是Huffman编码储存区，n个字节存编码代表的无符号数，n是半个字节的时候按一个字节计，n由上述存读取方式决定；4个字节存code的长度，然后存code。
- 编码后的文件流，4个字节存编码代表的无符号数，4个字节存code长度，code存成多长取上整（按8、16、24、32），高位补0。

压缩部分流程如下：

- 判断用户输入文件路径是否正确；
- 扫描文件，得出各字符频率，储存于map<uint,uint>中；
- 根据字符频率散列表构建堆，再根据此堆构建Huffman树；
- 用深度优先搜索对策将Huffman树进行编码；
- 第二遍扫描，根据压缩文件的设计写入压缩文件。

解压部分流程如下：

- 判断用户输入文件路径是否正确；
- 查看前两个字节，判断文件是否被压缩；

- 如果确实被压缩，先预处理文件头，即根据文件头信息建立Huffman编码与其所对应的字符无符号数映射表。
- 扫描密码流，查表得出原字符。
- 写入解压后的新文件。

# 关键代码讲解

堆的插入函数：把新插入的元素放到堆的最后，可能导致堆的性质被破坏，所以要进行堆化操作。

```
bool insert(HuffmanNode* node) {
    if (ptr < capacity) {
        this->e_a[ptr + 1] = node;
        for (uint i = ptr + 1; i / 2 > 0 && e_a[i]->w < e_a[i / 2]->w; i >>= 1) {
            swap(e_a[i], e_a[i / 2]);
        }
        this->ptr++;
        return true;
    }
    return false;
}
```

堆的删除函数：把最后一个元素放到第一个元素的位置，再堆化。

```
HuffmanNode* pop() {
    HuffmanNode* returnvalue = e_a[1];
    e_a[1] = e_a[ptr];
    e_a[ptr] = nullptr;
    ptr--;
    heapify();
    return returnvalue;
}
```

堆的堆化操作：向上调整，恢复堆的性质。

```
void heapify() {
    for (uint i = ptr / 2; i >= 1; i--) {
        while (1) {
            uint max = i;
            if (i * 2 <= ptr && e_a[i]->w > e_a[i * 2]->w)max = i * 2;
            if (i * 2 + 1 <= ptr && e_a[max]->w > e_a[i * 2 + 1]->w)max = i * 2 + 1;
            if (max == i)break;
            swap(e_a[i], e_a[max]);
            i = max;
        }
    }
}
```

多叉树的深度优先遍历得出Huffman编码：

```
void encode_helper(HuffmanNode* node) {
    uint fork_num = node->fork_num;
    for (uint i = 0; i < fork_num; i++) {
        if (node->child[i]) {
            if (fork_num <= 10) {
                node->child[i]->code = node->code + to_string(i);
            }
            else {
                if (i < 10) {
                    node->child[i]->code = node->code + to_string(i);
                }
                else {
                    node->child[i]->code = node->code + (char)(i - 10 + 'A');
                }
            }
            encode_helper(node->child[i]);
```

```
        }
    }
    return;
}
void insert_encoded_leaf_to_map(HuffmanNode* node) {
    if (node) {
        if (node->child_num == 0) {
            if (!node->is_virtual) {
                node->Regularize();
                code_map.insert(pair<uint, string>(node->name, node->code));
            }
            return;
        }
        else {
            for (uint i = 0; i < node->fork_num; i++) {
                insert_encoded_leaf_to_map(node->child[i]);
            }
        }
    }
    return;
}
void encode(HuffmanNode* node) {
    encode_helper(node);
    insert_encoded_leaf_to_map(node);
}
```

# 调试分析

经过助教线下验收，对大部分文件都能够进行压缩，压缩的效率视文件特性而定。

## 时间复杂度

### Heap类

一个包含 $n$ 个节点的完全二叉树的高度不会超过 $\log_2 n$ 。堆化的过程是顺着节点所在路径比较交换的,所以堆化的时间复杂度跟树的高度成正比，即 $O(\log n)$ 。插入数据和删除堆顶元素的主要逻辑就是堆化，所以往堆中插入一个元素和删除堆顶元素的时间复杂度都是 $O(\log n)$ 。

通过连续插入 $n$ 个元素的方式来建立堆，则建立堆的时间复杂度是 $O(n \log n)$。

### HuffmanTree类

对 $n$ 个节点的Huffman树进行编码（深度优先遍历），时间复杂度是 $O(n)$。其他操作如保存树为html代码的时间复杂度也是 $O(n)$。

## 空间复杂度

设读取位宽为 $k$ bit。

### Heap类

最多有 $2^k$ 种字符需要被统计频率，所以空间复杂度是 $O(2^k)$。

### HuffmanTree类

最多有 $2^k$ 种字符需要被记录编码，所以空间复杂度是 $O(2^k)$。

## 实验中遇到的问题

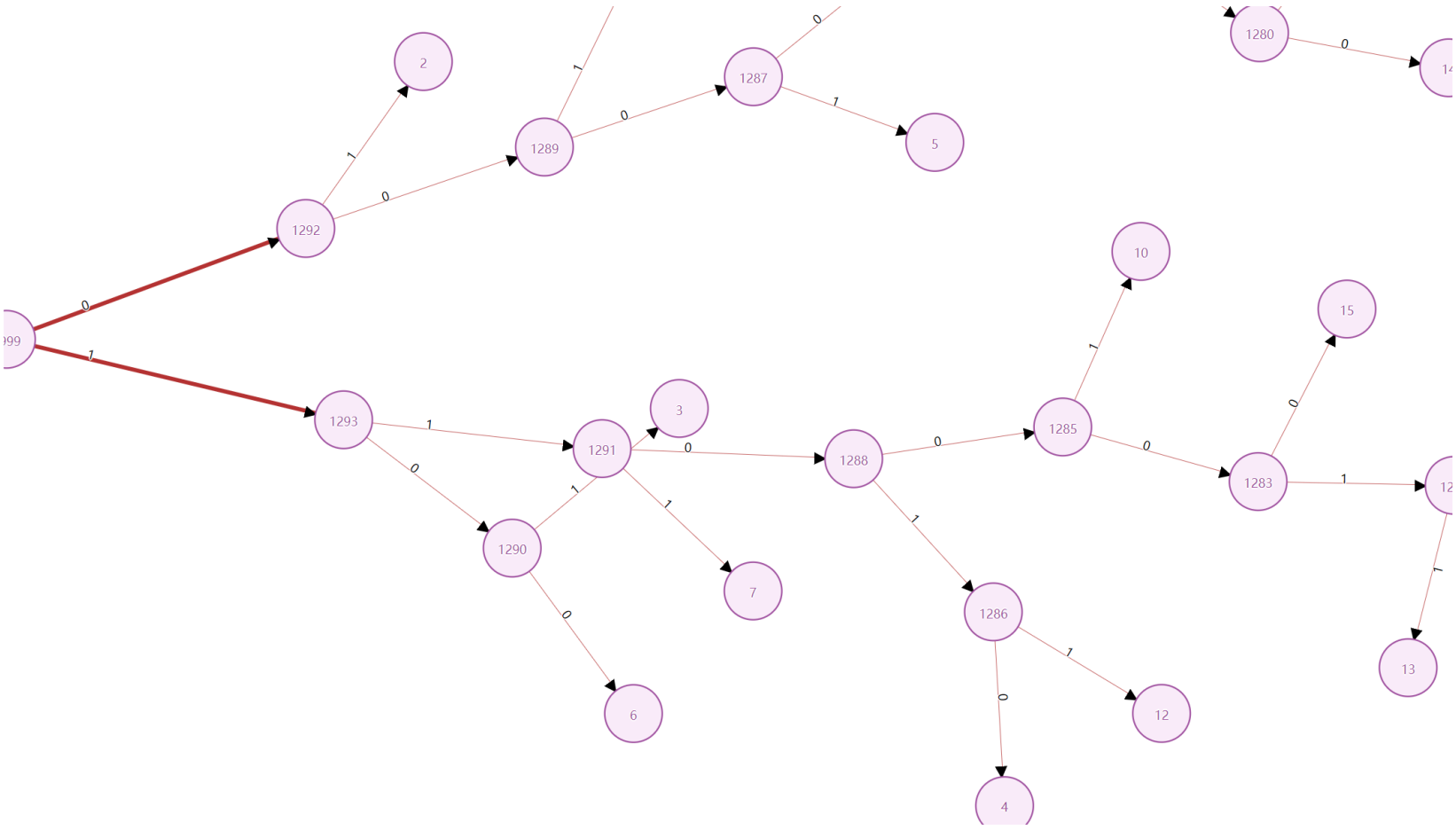
多叉Huffman树没有明确定义，按照原2叉树的方法进行则会导致根节点不满，从而不是最优树。所以要根据有效节点数和叉数进行调整，向树中插入若干权重为0的虚拟节点可以解决。

# 代码测试

经过对助教如下文件的测试：

```
civil_code.txt
huffman_test_files.zip
lena.png
model_show.mp4
to_my_future.wav
```

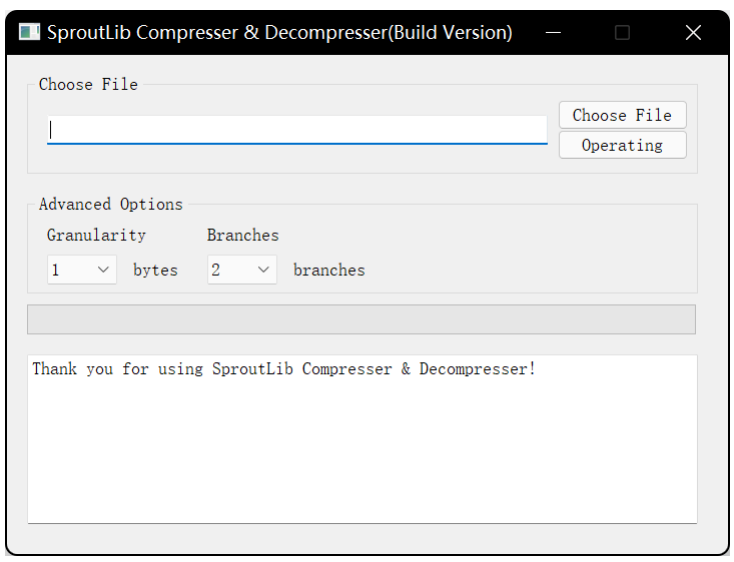
本实验的多字节、多叉树全部通过了验收，同时通过html展示所用Huffman树也正常工作。




# 实验总结

在本实验中，我学习了使用 Huffman 编码实现压缩/解压器，学习了设计堆、Huffman 树等数据结构，加深了对文件编码的理解。

我完成了实验的所有要求的基础部分，在发散部分，我做了以下工作：①完成了多字节情况；②完成了所有多叉树情况；③设计了图形化交互界面；④利用html展示压缩文件生成的Huffman树。




# 附录：文件清单

```
bitqueue.cpp
bitqueue.h
d3.v3.min.js
head.txt
heap.cpp
```

```
heap.h
huffmannode.cpp
huffmannode.h
huffmantree.cpp
huffmantree.h
main.cpp
mainwindow.cpp
mainwindow.h
mainwindow.ui
SproutLib.pro
tail.txt
ui_mainwindow.h
```