

文件压缩大作业报告

计算机 2 班

2252941 杨瑞灵

完成日期：2023 年 11 月 4 日

目录

1	设计思路与功能描述	3
1.1	设计思路	3
1.1.1	压缩方法: Huffman 树	3
1.1.2	程序功能划分	6
1.1.3	数据结构	6
1.2	功能描述	8
1.2.1	zip & unzip	8
1.2.2	主要函数功能	9
2	问题及解决方法	12
2.1	问题一: \r\n	12
2.2	一些小细节: priority_queue 的使用	14
3	心得体会	15
3.1	11 月 4 日	15
4	源代码	17
4.1	main.app	17
4.2	huffman.h	18
4.3	huffman.app	22

1 设计思路与功能描述

1.1 设计思路

1.1.1 压缩方法：Huffman 树

选择原因：由于数据结构刚刚讲完 Huffman 树，加上资料说 Huffman 可以压缩 20%-90% 压缩效率较高。

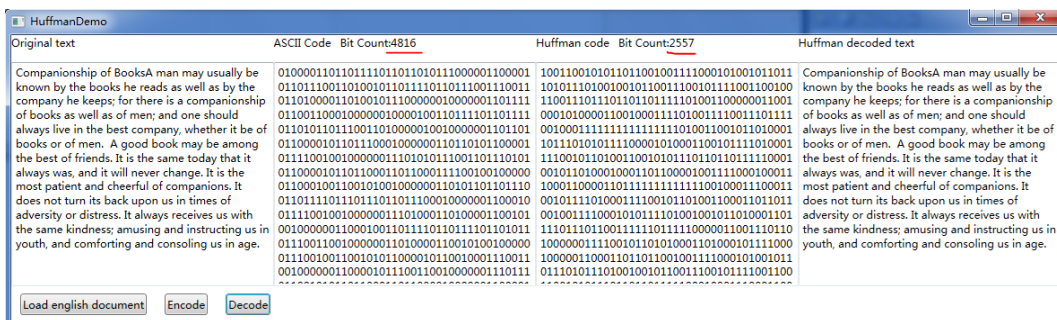


图 1: Huffman 压缩效率

方法介绍：

- 赫夫曼 (Huffman) 树，又称最优树，是一类带权路径长度最短的树。
 - 首先给出路径和路径长度的概念。从树中一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支。已复制到粘贴板的路径长度是从树根到每一结点的路径长度之和。完全二叉树就是这种路径长度最短的二叉树。
 - 若将上述概念推广到一般情况，考虑带权的结点。结点的带权路径长度为从该结点到树根之间的路径长度与结点上权的乘积。树的带权路径长度为树中所有叶子结点的带权路径长度之和，通常记作

$$WPL = \sum_{k=1}^n w_k l_k$$

- 假设有 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 。试构造一棵有 n 个叶子结点的二叉树，每个叶子结点带权为 w 。则其中带权路径长度 WPL 最小的二叉树称做最优二叉树或赫夫曼树。

例如,图 6.22 中的 3 棵二叉树,都有 4 个叶子结点 a、b、c、d,分别带权 7、5、2、4,它们的带权路径长度分别为

$$(a) WPL = 7 \times 2 + 5 \times 2 + 2 \times 2 + 4 \times 2 = 36$$

$$(b) WPL = 7 \times 3 + 5 \times 3 + 2 \times 1 + 4 \times 2 = 46$$

$$(c) WPL = 7 \times 1 + 5 \times 2 + 2 \times 3 + 4 \times 3 = 35$$

其中以(c)树的为最小。可以验证,它恰为赫夫曼树,即其带权路径长度在所有带权为 7、5、2、4 的 4 个叶子结点的二叉树中居最小。

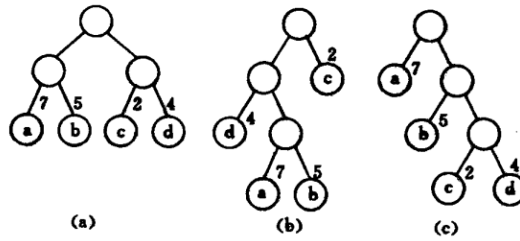


图 6.22 具有不同带权路径长度的二叉树

图 2: Huffman 树

- 赫夫曼算法

- (1) 根据给定的 n 个权值 $\{w_1, w_2, \dots, w_n\}$ 构成 n 棵二叉树的集合 $F = \{T_1, T_2, \dots, T_n\}$. 其中每棵二叉树 T 中只有一个带权为 w_i 的根结点, 其左右子树均空。

- (2) 在 F 中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树, 且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

- (3) 在 F 中删除这两棵树, 同时将新得到的二叉树加入 F 中。

- (1) 重复 (2) 和 (3), 直到 F 只含一棵树为止。这棵树便是赫夫曼树。

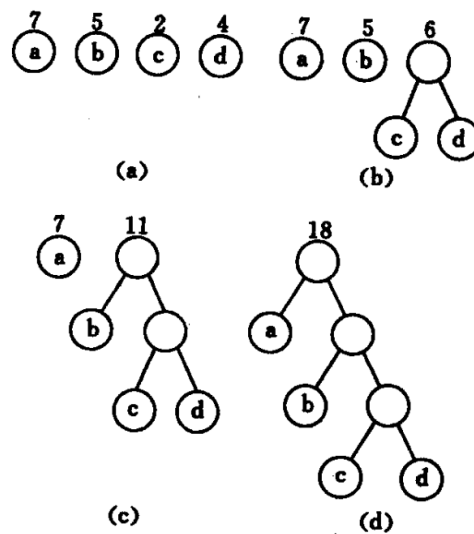


图 3: Huffman 树的建造过程

- 用 Huffman 树实现前缀编码
 - 要设计长短不等的编码, 必须是任一个字符的编码都不是另一个字符的编码的前缀, 这种编码称做前缀编码。
 - 可以利用二叉树来设计二进制的前缀编码。假设有一棵如图 6. 25 所示的二叉树, 其 4 个叶子结点分别表示 ABCD 这 4 个字符, 且约定左分支表示字符'0', 右分支表示字符'1', 则可以从根结点到叶子结点的路径上分支字符组成的字符串作为该叶子结点字符的编码。这样得到的必为二进制前缀编码。如由图所得 A、B、C、D 的二进制前缀编码分别为 0、10、110 和 111。

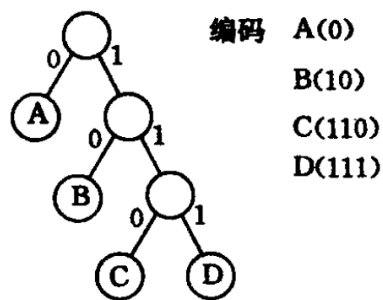


图 4: 前缀编码

1.1.2 程序功能划分

main: 实现调用各个功能
Huffman: 封装类

1.1.3 数据结构

- 逻辑结构: 二叉树
- 物理结构: 链表 (其实用数组会简单一点)

数据结构代码

```

1 typedef struct HuffmanNode {
2     char data;
3     int frequency;
4     HuffmanNode* left;
5     HuffmanNode* right;
6     HuffmanNode(char d, int freq, HuffmanNode* l =
7         nullptr, HuffmanNode* r = nullptr) : data(d),
8         frequency(freq), left(l), right(r) {}
9     bool operator>(const HuffmanNode& other) const {
10         return frequency > other.frequency;
11     }

```

```

10 }HuffmanNode;
11
12 typedef struct HuffmanTree {
13 private:
14     HuffmanNode* root;
15     std::map<char, std::string> codes;
16     void createHuffmanTree(const std::string text);
17     void buildHuffmanCodes(HuffmanNode* p, std::string
        code = "");
18     void buildHuffmanCodes(std::string& text);
19     void TurncharFile(const std::string binaryData,
        std::string& charData);
20     void TurnbinaryFile(const std::string charData,
        std::string& binaryData);
21     void Delete_each(HuffmanNode* p);
22 public:
23     HuffmanTree(char data = '\\0', int freq = 0)
24     {
25         root = new HuffmanNode(data, freq);
26     }
27     void encodeText(std::string text, std::string&
        encodedText);
28     void decodeText(std::string text, std::string&
        decodedText);
29     ~HuffmanTree()
30     {
31         Delete_each(root);
32     }
33 }HuffmanTree;

```

```

34
35 struct Zip_Uzip_file {
36 private:
37     const char* inputname;
38     const char* outputname;
39     const char* order;
40     std::string text;
41     std::string t_outputText, outputText;
42     // 建立HuffmanTree对象并生成outputText
43     HuffmanTree huffmanTree;
44     bool Openfile();//读取文件，成功返回1，失败返回0
45     bool Writefile();
46 public:
47     Zip_Uzip_file(const char* i, const char* o, const
48         char* b);
49 };

```

1.2 功能描述

1.2.1 zip & unzip

- 实现压缩，压缩比例为 65%
- 实现解压缩，解码后完全相同，无损压缩




 ser.log	2023/11/3 20:46	文本文档	10,073 KB
 uziptext.log	2023/11/4 15:31	文本文档	10,073 KB
 ziptext.log	2023/11/4 15:30	文本文档	6,556 KB

图 5:

	10000011
	0011
!	0100001001010000
#	0100001001010011
%	01000010010101
&	01000010010100010
(0110001
)	0110100
*	110
+	11101
,	00010101
-	100100
.	1010
/	11110
0	0101
1	01001
2	101100
3	01110
4	001011
5	100001
6	101110
7	010001
8	1011110
9	1000111
:	000100
;	0110110
=	010000101
?	1111100101
A	10000001
B	0100001000
C	00101011
D	100010010001
E	10000010
F	11111001001
G	0001011
H	01000011
I	10111110100

图 6: 部分编码

1.2.2 主要函数功能

- HuffmanNode 结构体

```

typedef struct HuffmanNode {
    char data;
    int frequency;
    HuffmanNode* left; //左孩子
    HuffmanNode* right; //右孩子
    HuffmanNode(char d, int freq, HuffmanNode* l = nullptr, HuffmanNode* r = nullptr) {
        //重定义操作符 >, 方便优先队列进行排序
        bool operator>(const HuffmanNode& other) const { ... }
    }
} HuffmanNode;

```

图 7: 节点结构体

• HuffmanTree 最优二叉树

```

typedef struct HuffmanTree {
private:
    HuffmanNode* root;
    std::map<char, std::string> codes;
    //创建HuffmanTree
    void createHuffmanTree(const std::string text);
    //根据HuffmanTree创建Huffman表
    void buildHuffmanCodes(HuffmanNode* p, std::string code = "");
    void buildHuffmanCodes(std::string& text);
    // "00101010" 转成char串
    void TurncharFile(const std::string binaryData, std::string& charData);
    //char的字符串转成"01000010"
    void TurnbinaryFile(const std::string charData, std::string& binaryData);
    //释放空间的函数
    void Delete_each(HuffmanNode* p);
public:
    HuffmanTree(char data = '\0', int freq = 0) { ... }
    //zip
    void encodeText(std::string text, std::string& encodedText);
    //unzip
    void decodeText(std::string text, std::string& decodedText);
    ~HuffmanTree() { ... }
} HuffmanTree;

```

图 8: 节点结构体

• Zip_Uzip_file 压缩和解压缩

```

struct Zip_Uzip_file {
private:
    //文件名
    const char* inputname;
    const char* outputname;
    const char* order;
    //文件内容存放在text中
    std::string text;
    //输出内容存放在outputText中
    std::string t_outputText, outputText;
    // 建立HuffmanTree对象并生成outputText
    HuffmanTree huffmanTree;
    bool Openfile() { ... }
    bool Writefile() { ... }
public:
    Zip_Uzip_file(const char* i, const char* o, const char* b);
};

```

图 9: 节点结构体

2 问题及解决方法

2.1 问题一：\r\n

- **问题描述：**如果没有处理 \t\n 那么应该是有 82 种字符，并且解压缩会和源文件有不一样的地方

- **问题解答：**

- 在 Windows 操作系统下读取文件的\underline{n} 会变成\underline{r}\uunderline{n}
- 而书写文件的\underline{n} 也会变成\underline{r}\uunderline{n}

要写入的字符	实际写入的字符（即\n被替换为\r\n后）	Ultraedit转换为DOS格式后的文件内的实际字符
\r	\r	\r\n
\n	\r\n	未提示转换为DOS文件
\r\n	\r\r\n	\r\n
\n\r	\r\n\r	未提示转换为DOS文件
\n\r\n	\r\n\r\n	\r\n\r\n
\n\r\n\r	\r\n\r\n\r	\r\n\r\n
\n\r\n\r\n	\r\n\r\n\r\n\r	\r\n\r\n\r\n\r\n
\r\r\n	\r\r\n\r\n	\r\n\r\n\r\n
\r\r\n\r\n	\r\r\n\r\n\r\n	\r\n\r\n\r\n\r\n
\n\r\r	\r\n\r\r	\r\n\r\n\r\n
\n\r\r\r	\r\n\r\r\r	\r\n\r\n\r\n\r\n\r\n

图 10: \r\n

- 也就是说如果不加处理那么读取一遍,再输出一遍就会由\underline{n} 变成\underline{r}\uunderline{r}\uunderline{n}
- 所以无论是 zip 还是 unzip 都需要把\underline{r}\uunderline{n} 中的\underline{r} 删掉。查找和删除的时间复杂度非常重要因为要进行频繁的搜索。

C++的string类提供了多个函数用于查找和删除字符。以下是一些常用的函数及其算法和时间复杂度：

1. `find(char c)`: 查找字符c在字符串中第一次出现的位置。算法使用线性搜索，时间复杂度为 $O(n)$ ，其中n是字符串的长度。
2. `rfind(char c)`: 查找字符c在字符串中最后一次出现的位置。算法使用线性搜索，时间复杂度为 $O(n)$ ，其中n是字符串的长度。
3. `find_first_of(string str)`: 查找字符串str中任意字符在原字符串中第一次出现的位置。算法使用线性搜索，时间复杂度为 $O(n*m)$ ，其中n是原字符串的长度，m是str的长度。
4. `find_last_of(string str)`: 查找字符串str中任意字符在原字符串中最后一次出现的位置。算法使用线性搜索，时间复杂度为 $O(n*m)$ ，其中n是原字符串的长度，m是str的长度。
5. `erase(int pos)`: 删除从位置pos开始的字符。算法将后面的字符向前移动，时间复杂度为 $O(n)$ ，其中n是字符串的长度。
6. `erase(int pos, int len)`: 删除从位置pos开始的len个字符。算法将后面的字符向前移动，时间复杂度为 $O(n)$ ，其中n是字符串的长度。

需要注意的是，以上函数的时间复杂度是基于字符串的长度，而不是字符的个数。因此，如果字符串很长，这些操作可能会比较耗时。在实际使用中，如果需要频繁进行查找和删除操作，可以考虑使用其他数据结构或算法来提高效率。

图 11: string 查找和删除函数

。虽然他们的时间复杂度都是 $O(n)$ 但是由于要查找很多次所以实际上时间复杂度是 $O(m * n)$ (m 是需要删除的字符个数)。所以有没有什么办法可以一次性找出所有要删除的字符呢。当然可以：

```

//除掉/t
//改进后
text.erase(std::remove(text.begin(), text.end(), '\r'), text.end());
//改进前
size_t cou;
int f = 0;
while ((cou = text.find('\r', f)) != std::string::npos)
{
    text.erase(cou, 1);
    f = cou + 1;
}

```

图 12: 改进查找删除字符\r

- 时间复杂度 $O(n)$ 。

2.2 一些小细节：priority_queue 的使用

- 优先队列比 queue 多排序功能，用来构造最优二叉树非常方便。在配合一下操作符 > 的重定义。

```

std::priority_queue<HuffmanNode, std::vector<HuffmanNode>, std::greater<HuffmanNode>> pqueue;
for (auto& entry : Map) {
    pqueue.push(HuffmanNode(entry.first, entry.second));
}

//重定义操作符 > , 方便优先队列进行排序
bool operator>(const HuffmanNode& other) const {
    return frequency > other.frequency;
}

```

图 13: priority_queue

3 心得体会

3.1 11 月 4 日

- 一天干掉大作业，愉快愉快



- 现在是正文：

- 尝试着去封装类，简化函数，不知道算不算成功，感觉 HuffmanNode 和 HuffmanTree 是不是可以设置成只对 Zip_Uzip_file 可获取，对外 private。
- 熟悉了 string 类的一些函数，以及优先队列等容器的使用，还有运算符重构。
- 了解了字符编码的一些规则
- 最后的压缩比例其实不算高甚至没有到 50%。
- 分析：

1. Huffman 受限于字符频率的分布，如果大部分字符出现频率相

近那么效果就不是很明显。像这种日志就不适用

LZ 系列字典压缩算法 适合于大量重复字符串，这可能是为什么这次的作业用 LZ 会明显好过 Huffman 的原因

2. 由于叶子较多 Huffman 树的查找很费时间,我的压缩时间是五秒,解码时间更短,是两秒。

3. 就实际生活中 Huffman 虽然常用但一般不会单独使用，都会配合 LZ 和其他编码方式一同使用。

4 源代码

4.1 main.app

```
1  #include <iostream>
2  #include <fstream>
3  #include <chrono>
4  #include <map>
5  #include "huffman.h"
6  using namespace std;
7
8  int main(int argc, char* argv[])
9  {
10     //技术开始
11     auto starttime = chrono::high_resolution_clock::now();
12     cout << "Zipper 0.001! Author: root" << endl;
13     if (argc != 4) {
14         cerr << "Please make sure the number of parameters
15             is correct." << endl;
16         return -1;
17     }
18     if (strcmp(argv[3], "zip") && strcmp(argv[3], "uzip"))
19     {
20         cerr << "Unknown parameter!\nCommand list:\nzip"
21             << endl;
22         return -1;
23     }
24
25     Zip_Uzip_file File(argv[1], argv[2], argv[3]);
```

```

23
24 //记录时间
25 auto endtime =
    std::chrono::high_resolution_clock::now();
26 auto duration =
    std::chrono::duration_cast<chrono::microseconds>(endtime
    - starttime);
27 cout << "运行时间: " << double(duration.count()) *
    chrono::microseconds::period::num /
    chrono::microseconds::period::den << " 秒" <<
    std::endl;
28 return 0;
29 }

```

4.2 huffman.h

```

1 #pragma once
2 #ifndef HUFFMAN_H
3 #define HUFFMAN_H
4 #include <iostream>
5 #include <fstream>
6 #include <map>
7 //using namespace std;
8 typedef struct HuffmanNode {
9     char data;
10    int frequency;
11    HuffmanNode* left; //左孩子
12    HuffmanNode* right; //右孩子
13    HuffmanNode(char d, int freq, HuffmanNode* l =

```

```

        nullptr, HuffmanNode* r = nullptr) : data(d),
        frequency(freq), left(l), right(r) {}
14 //重定义操作符 > ,方便优先队列进行排序
15 bool operator>(const HuffmanNode& other) const {
16     return frequency > other.frequency;
17 }
18 }HuffmanNode;
19
20 typedef struct HuffmanTree {
21 private:
22     HuffmanNode* root;
23     std::map<char, std::string> codes;
24     //创建HuffmanTree
25     void createHuffmanTree(const std::string text);
26     //根据HuffmanTree创建Huffman表
27     void buildHuffmanCodes(HuffmanNode* p, std::string
        code = "");
28     void buildHuffmanCodes(std::string& text);
29     //"00101010"转成char串
30     void TurncharFile(const std::string binaryData,
        std::string& charData);
31     //char的字符串转成"01000010"
32     void TurnbinaryFile(const std::string charData,
        std::string& binaryData);
33     //释放空间的函数
34     void Delete_each(HuffmanNode* p);
35 public:
36     HuffmanTree(char data = '\0', int freq = 0)
37     {

```

```

38         root = new HuffmanNode(data, freq);
39     }
40     //zip
41     void encodeText(std::string text, std::string&
        encodedText);
42     //unzip
43     void decodeText(std::string text, std::string&
        decodedText);
44     ~HuffmanTree()
45     {
46         Delete_each(root);
47     }
48 }HuffmanTree;
49
50 struct Zip_Uzip_file {
51 private:
52     //文件名
53     const char* inputname;
54     const char* outputname;
55     const char* order;
56     //文件内容存放在text中
57     std::string text;
58     //输出内容存放在outputText中
59     std::string t_outputText, outputText;
60     // 建立HuffmanTree对象并生成outputText
61     HuffmanTree huffmanTree;
62     bool Openfile()//读取文件，成功返回1，失败返回0
63     {
64         std::ifstream fin(inputname, std::ios::binary); //

```

```

        以二进制方式打开文件
65         if (!fin)
66             return 0;
67         std::istreambuf_iterator<char> beg(fin), end; //
            设置两个文件指针，指向开始和结束，以
            char(一字节) 为步长
68         std::string t(beg, end); // 将文件全部读入 string
            字符串
69         text = t;
70         fin.close(); //
            操作完文件后关闭文件句柄是一个好习惯
71         return 1;
72     }
73     bool Writefile()
74     {
75         //文件输出
76         std::ofstream fout(outputname); // 打开输出文件
77         if (!fout)
78             return 0;
79         fout << outputText; // 直接将操作好的字符串进行输出
80         fout.close();
81         return 1;
82     }
83     public:
84         Zip_Uzip_file(const char* i, const char* o, const
            char* b);
85 };
86
87 #endif

```

4.3 huffman.app

```
1     #include <vector>
2     #include <queue>
3     #include <map>
4     #include <iostream>
5     #include <bitset>
6     #include "huffman.h"
7
8     // 创建 HuffmanTree
9     void HuffmanTree::createHuffmanTree(const std::string text)
10    {
11        std::map<char, int> Map;
12        for (int i = 0; i < text.size(); i++) {
13            Map[text[i]]++;
14        }
15
16        std::priority_queue<HuffmanNode,
17                            std::vector<HuffmanNode>,
18                            std::greater<HuffmanNode>> pqueue;
19        for (auto& entry : Map) {
20            pqueue.push(HuffmanNode(entry.first,
21                                    entry.second));
22        }
23
24        while (pqueue.size() > 1) {
25            HuffmanNode* lchild = new
26                HuffmanNode(pqueue.top().data,
27                            pqueue.top().frequency, pqueue.top().left,
```

```

        pqueue.top().right);
23     pqueue.pop();
24     HuffmanNode* rchild = new
        HuffmanNode(pqueue.top().data,
        pqueue.top().frequency, pqueue.top().left,
        pqueue.top().right);
25     pqueue.pop();
26     HuffmanNode* parent = new HuffmanNode('\0',
        lchild->frequency + rchild->frequency, lchild,
        rchild);
27     pqueue.push(*parent);
28 }
29 root->frequency = pqueue.top().frequency;
30 //huffmanTree.root = new HuffmanNode('\0',
    pqueue.top().frequency);
31 root->left = pqueue.top().left;
32 root->right = pqueue.top().right;
33 }
34
35 //根据HuffmanTree创建Huffman表map codes
36 void HuffmanTree::buildHuffmanCodes(HuffmanNode* p,
    std::string code)
37 {
38     if (p == nullptr)
39         return;
40     if (!p->left && !p->right) {
41         codes[p->data] = code;
42     }
43     else {

```

```

44         buildHuffmanCodes(p->left, code + "0");
45         buildHuffmanCodes(p->right, code + "1");
46     }
47     return;
48 }
49 void HuffmanTree::buildHuffmanCodes(std::string &text)
50 {
51     //除掉\r\n中的\r
52     size_t cou;
53     int f = 0;
54     while ((cou = text.find("\r\n")) != std::string::npos){
55         text.erase(cou, 1);
56         f = cou;
57     }
58     std::string s;
59     while (1) {
60         int cn = text.find_first_of('\0');
61         if (cn == 0) {
62             text.erase(0, 1);
63             break;
64         }
65         std::string s = text.substr(1, cn - 1);
66         codes[text[0]] = s;
67         text.erase(0, cn + 1);
68     }
69 }
70
71 //zip
72 void HuffmanTree::encodeText(std::string text, std::string&

```



```

encodedText)
73 {
74     //除掉/t
75
76     //改进后
77     text.erase(std::remove(text.begin(), text.end(),
78         '\r'), text.end());
79     ////改进前
80     //size_t cou;
81     //int f = 0;
82     //while ((cou = text.find('\r', f)) !=
83         std::string::npos)
84     //{
85         //text.erase(cou, 1);
86         //f = cou + 1;
87     //}
88
89     //创建Huffman树
90     std::cout << "1" << std::endl;
91     createHuffmanTree(text);////
92     //创建Huffman编码表
93     buildHuffmanCodes(root);
94     std::cout << "1" << std::endl;
95     //用huffman编码编写文件
96     std::string t_outputText;/////
97     for (char i : text) {
98         t_outputText += codes[i];
99     }
100     //把"01010110100000"写成char

```

```

99     TurncharFile(t_outputText, encodedText);
100 }
101
102 //uzip
103 void HuffmanTree::decodeText(std::string text,
104                               std::string& decodedText)
105 {
106     //建立Huffman表
107     buildHuffmanCodes(text);
108     //把char的字符串翻译成"0010101001"字符串
109     std::string t_outputText;
110     TurnbinaryFile(text, t_outputText);
111     //建树
112     for (auto i : codes) {
113         HuffmanNode* p = root;
114         for (char j : i.second) {
115             if (j == '0') {
116                 if (!p->left){
117                     p->left = new HuffmanNode('\0', 0);
118                 }
119                 p = p->left;
120             }
121             else {
122                 if (!p->right) {
123                     p->right = new HuffmanNode('\0', 0);
124                 }
125                 p = p->right;
126             }
127         }
128     }

```

```

127         p->data = i.first;
128     }
129     HuffmanNode* p = root;
130     for (char i : t_outputText) {
131         if (i == '0') {
132             p = p->left;
133         }
134         else {
135             p = p->right;
136         }
137         if (!p->left && !p->right) {
138             decodedText += p->data;
139             p = root;
140         }
141     }
142 }
143
144 // "00101010" 转成char串
145 void HuffmanTree::TurncharFile(const std::string
    binaryData, std::string &charData) {
146     int cou = 0;
147     unsigned char ch = '\0';
148     // 输出Huffman编码表
149     for (auto i : codes) {
150         std::cerr << i.first << "    " << i.second <<
            std::endl;
151         charData += i.first;
152         charData += (i.second + '\0');
153     }

```

```

154     charData += '\\0';//\\0标志结束
155     //输出编码后的文本
156     for (char i : binaryData) {
157         ch <<= 1;
158         ch += i - '0';
159         cou++;
160         if (cou == 8) {
161             charData += ch;
162             cou = 0;
163             ch &= 0;
164         }
165     }
166     //处理最后不足文本
167     ch <<= (8 - cou);//只有前cou位有效
168     charData += ch;
169     charData += char(8 - cou);//最后一个字节记录无效位数
170 }
171
172 //char的字符串转成"01000010"
173 void HuffmanTree::TurnbinaryFile(const std::string
    charData, std::string& binaryData)
174 {
175     int cou = charData[charData.size() - 1];
176     for (char i : charData) {
177         for (int bit = 7; bit >= 0; --bit) {
178             binaryData += ((i >> bit) & 1) ? "1" : "0";
179         }
180     }
181     binaryData.erase(binaryData.size() - cou - 8, cou + 8);

```

```

182 }
183
184 //释放空间的函数
185 void HuffmanTree::Delete_each(HuffmanNode* p) {
186     if (p) {
187         Delete_each(p->left);
188         Delete_each(p->right);
189         delete p;
190     }
191 }
192
193 //压缩函数
194 Zip_Uzip_file::Zip_Uzip_file(const char* i, const char* o,
195     const char* b) :inputname(i), outputname(o), order(b)
196 {
197     if (!Openfile()) {
198         std::cerr << "Can not open the input file!" <<
199             std::endl; // 输出错误信息并退出
200         exit;
201     }
202     if (!strcmp(order, "zip"))//zip
203         huffmanTree.encodeText(text, outputText);
204     else if (!strcmp(order, "uzip"))//uzip
205         huffmanTree.decodeText(text, outputText);
206     //huffmanTree.Delete();//手动delet一下防止析构函数出各种问题
207     else
208         return;
209     if (!Writefile()) {
210         std::cerr << "Can not open the output file!" <<

```

```
        std::endl;
209     exit;
210 }
211 else {
212     std::cout << "Complete!" << std::endl;
213 }
214 }
```