

1.题目名称

Huffman编码与解码

2.代码行数

228行

3.算法思想

1.创建Huffman树

每次选取最小的两个节点，从vector中拿出来，并开一个新的节点，左右孩子节点指向拿出的两个节点，将该节点插入vector。

2.编码

从根节点开始，如果往左子树走则编码左移并低位置0，反之编码左移置1。找到目标节点则返回编码。

这里为了移位方便，我初始编码为1，后续转二进制会对之进行处理。

3.编码以比特位写入文件

每次8个bit写入文件，如最后剩余bit数不足8个则左移后写入文件。

4.解码

按照获取的编码对Huffman树进行遍历，找到叶子节点则退出，从根节点重新开始遍历。

4.主要/核心函数分析

CreateTree

```
1 void Huffman::CreateTree(vector<CharSum*> cs){
2     CharSum *temp;
3     while(cs.size()>1){           //每次寻找两个最小的合成一个节点
4         sort(cs.begin(),cs.end(),cmp);
5         temp=new CharSum;
6         temp->left=cs.front();
7         cs.erase(cs.begin());
8         temp->right=cs.front();
9         cs.erase(cs.begin());
10        temp->ch=temp->left->ch+" "+temp->right->ch;
11        temp->Sum=temp->left->Sum+temp->right->Sum;
12        cs.push_back(temp);
13    }
14    root=cs.front();
15    cs.pop_back();
16    return;
17 }
```

通过循环不断地取出字符频率统计数组中频率最小的两个节点，合并成一个新的节点，然后将新节点加入字符频率统计数组。重复这个过程，直到字符频率统计数组中只剩下一个节点，这个节点即为赫夫曼树的根节点。

Huffman_Encode

```
1 void Huffman::Huffman_Encode(CharSum *head , CharSum **p ,int ct) {
2     if((*p)->ch==head->ch){
3         (*p)->Encoding=ct;
4         return ;
5     }
6
7     if(head->left!= nullptr){
8         ct<<=1;    //置0
9         Huffman_Encode(head->left,p,ct);
10        ct>>=1;
11    }
12
13    if(head->right!= nullptr){
14        ct=(ct<<1)|1;    //置1
15        Huffman_Encode(head->right,p,ct);
16        ct>>=1;
17    }
18 }
```

从根节点开始，如果往左子树走则编码左移并低位置0，反之编码左移置1。找到目标节点则返回编码。
这里为了移位方便，我初始编码为1，后续转二进制会对之进行处理。

writeEncodedTextToFile

```
1 void Huffman::writeEncodedTextToFile(vector<CharSum *> cs) {
2     string encodedText="";
3     for(int i=0;i<cs.size();i++){
4         encodedText+= To_Binary(cs[i]->Encoding);
5     }
6
7     fstream fileout;
8     fileout.open("../code.dat",ios::out|ios::binary);
9     if(!fileout.is_open()){
10        cout<<"Write Error"<<endl;
11        exit(0);
12    }
13
14    bitset<8> bits;
15    for (char c : encodedText) {    //8位一组写入文件
16        bits <= 1;
17        if (c == '1') {
18            bits |= 1;
19        }
20        if (bits.size() == 8) {
21            fileout.put(static_cast<unsigned char>(bits.to_ulong()));
22            bits.reset();
23        }
24    }
25    if (bits.size() > 0) {
26        bits <= (8 - bits.size());
27        fileout.put(static_cast<unsigned char>(bits.to_ulong()));
28    }
29    fileout.close();
30 }
```

以二进制模式写入文件，对编码按每8个bit写入文件，如最后剩余bit数不足8个则左移后写入文件。

decodeText

```
1 void Huffman::decodeText(const string& encodedText) {
2     string decodedText;
3     CharSum* current = root;
4     vector<int> Count;
5     vector<int> Huffman;
6     for (char c : encodedText) { //照着遍历,找到节点则解码成功
7         if (c == '0') {
8             current = current->left;
9         } else if (c == '1') {
10            current = current->right;
11        }
12
13        if (current->left == nullptr && current->right == nullptr) {
14            decodedText += current->ch;
15            Count.push_back(current->Sum);
16            Huffman.push_back(current->Encoding);
17            current = root;
18        }
19    }
20
21    ofstream fileout;
22    fileout.open("../decode.txt", ios::out);
23    if(!fileout.is_open()){
24        cout<<"Write Decode.txt Error"<<endl;
25        exit(0);
26    }
27    for(auto it : decodedText){
28        char kt=it;
29        if (kt >= 0 && kt <= 32) {
30            fileout << (int)kt <<"\t\t"<< Count.front()<<"\t\t"
<<To_Binary(Huffman.front())<<std::endl;
31        } else {
32            fileout << kt <<"\t\t"<< Count.front()<<"\t\t"
<<To_Binary(Huffman.front())<<std::endl;
33        }
34        Count.erase(Count.begin());
35        Huffman.erase(Huffman.begin());
36    }
37    fileout.close();
38    return ;
39 }
```

按照获取的编码对Huffman树进行遍历，如果为0则往左子树遍历，反之则往右子树遍历，找到叶子节点则退出，从根节点重新开始遍历。

5.测试数据(规模,测试次数)

规模:不少于5000字符的英文文章

测试次数:1

测试用例:见测试文件

6.运行结果

Huffman.txt

1	2	23	001001110
2	10	161	001000
3	13	161	1111111
4	32	1827	101
5	%	2	1111110011001
6	'	1	11111100111010
7	(19	1111110101
8)	21	1111110110
9	,	110	1101000
10	-	49	10010001
11	.	104	1001001
12	/	1	11111100111011
13	0	2	1111110011100
14	1	6	11010010110
15	2	3	1111110111101
16	3	1	11111100110101
17	4	1	11111100110100
18	9	2	1111110011000
19	:	1	11111101111000
20	;	5	111111011100
21	A	22	001001101
22	B	4	111111001111
23	C	21	001001010
24	D	25	110100100
25	E	7	11010010111
26	F	13	1101001010
27	G	15	1101001101
28	H	6	111111011111
29	I	24	100100000
30	L	1	11111101111001
31	M	22	001001011
32	N	12	0010011111
33	O	14	1101001100
34	P	12	1001000011
35	R	5	111111011101
36	S	12	1001000010
37	T	32	111111000
38	U	8	11111100100
39	V	2	1111110011011
40	W	11	0010011110
41	a	899	1100
42	b	135	1111000
43	c	416	01101
44	d	356	00101
45	e	1295	000
46	f	233	110101
47	g	217	100101
48	h	369	01100
49	i	856	1000
50	j	22	001001100
51	k	42	00100100
52	l	454	10011
53	m	271	110111

54	n	750	0101
55	o	829	0111
56	p	275	111101
57	q	19	1111110100
58	r	703	0100
59	s	696	0011
60	t	1080	1110
61	u	283	111110
62	v	132	1101101
63	w	123	1101100
64	x	31	110100111
65	y	138	1111001
66	z	8	11111100101
67			

code.dat

由于是比特位写入，因此这里给出编码，不给出文件内容。

```
1 00100111000100011111110111111001100111111100111010111111010111111011011010
001001000110010011111110011101111111001110011010010110111111011110111111001
1010111111100110100111111001100011111101111000111111011100001001101111110011
110010010101101001001101001011111010010101101001101111110111110010000011111
1011110010010010110010011111110100110010010000111111101110110010000101111110
00111111001001111110011011001001111011001111000011010010100011010110010101100
1000001001100001001001001111011101010111111101111110100010000111110111110110
11011101100110100111111100111111100101
```

decode.txt

1	2	23	001001110
2	10	161	001000
3	13	161	1111111
4	32	1827	101
5	%	2	1111110011001
6	'	1	11111100111010
7	(19	1111110101
8)	21	1111110110
9	,	110	1101000
10	-	49	10010001
11	.	104	1001001
12	/	1	11111100111011
13	0	2	1111110011100
14	1	6	11010010110
15	2	3	1111110111101
16	3	1	11111100110101
17	4	1	11111100110100
18	9	2	1111110011000
19	:	1	11111101111000
20	;	5	111111011100
21	A	22	001001101
22	B	4	111111001111
23	C	21	001001010
24	D	25	110100100
25	E	7	11010010111
26	F	13	1101001010

27	G	15	1101001101
28	H	6	111111011111
29	I	24	100100000
30	L	1	11111101111001
31	M	22	001001011
32	N	12	0010011111
33	O	14	1101001100
34	P	12	1001000011
35	R	5	111111011101
36	S	12	1001000010
37	T	32	111111000
38	U	8	11111100100
39	V	2	1111110011011
40	W	11	0010011110
41	a	899	1100
42	b	135	1111000
43	c	416	01101
44	d	356	00101
45	e	1295	000
46	f	233	110101
47	g	217	100101
48	h	369	01100
49	i	856	1000
50	j	22	001001100
51	k	42	00100100
52	l	454	10011
53	m	271	110111
54	n	750	0101
55	o	829	0111
56	p	275	111101
57	q	19	1111110100
58	r	703	0100
59	s	696	0011
60	t	1080	1110
61	u	283	111110
62	v	132	1101101
63	w	123	1101100
64	x	31	110100111
65	y	138	1111001
66	z	8	11111100101
67			

7.时间复杂度分析

`CreateTree` 循环内主要复杂度取决于排序的操作，时间复杂度为 $O(n^2 \log n)$

`Huffman_Encode` 遍历节点时间复杂度为 $O(n)$

`writeEncodedTextToFile`：

1. 将每个字符的编码转换为二进制字符串： $O(n)$
2. 将这些二进制字符串连接成一个大的二进制字符串： $O(n)$
3. 将这个大的二进制字符串分成 8 位一组： $O(n)$
4. 将这些 8 位一组的二进制字符串写入文件： $O(n)$

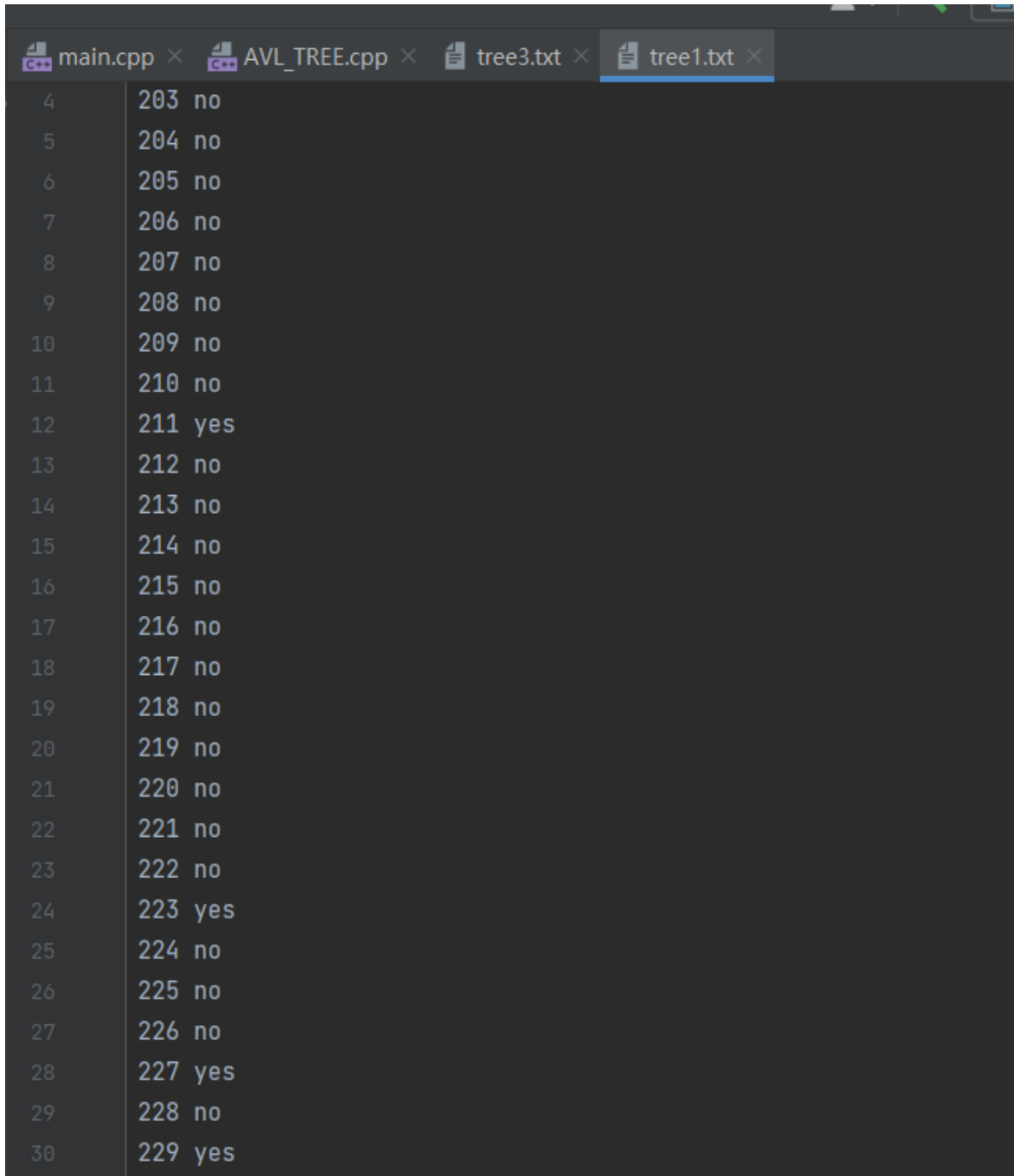
因此，整体上，这段代码的时间复杂度为 $O(n)$ 。

decodeText:

1. 遍历编码文本并解码字符: $O(n)$
2. 将叶节点的权重和编码添加到两个向量中: $O(n)$
3. 将解码文本和叶节点的权重和编码写入文件中的过程: $O(n)$

因此程序复杂度为 $O(n^2 \log n)$ 。

8.结果截屏图片



```
4      203 no
5      204 no
6      205 no
7      206 no
8      207 no
9      208 no
10     209 no
11     210 no
12     211 yes
13     212 no
14     213 no
15     214 no
16     215 no
17     216 no
18     217 no
19     218 no
20     219 no
21     220 no
22     221 no
23     222 no
24     223 yes
25     224 no
26     225 no
27     226 no
28     227 yes
29     228 no
30     229 yes
```

9.心得体会

通过该题,对Huffman树的基本性质有了更深入的了解, 同时,对Huffman在编码以及解码这一方面的应用以及实现更为熟悉了。