

# 《数据结构》上机报告

2020 年 11 月 22 日

姓名： 王上游      学号： 1850767    班级：    19 计科 2 班      得分： \_\_\_\_\_

实验题目	哈夫曼编码和译码	
实验目的	理解最优二叉树，即哈夫曼树(Huffman tree)的概念，熟悉它的构造过程。 实现对 ASCII 字符文本进行 Huffman 压缩，并且能够进行解压。	
问题描述	通讯理论中的一个基本问题是，如何在尽可能低的成本下，以尽可能高的速度，尽可能忠实地实现信息在空间和时间上的复制与转移。在现代通讯技术中，无论采用电、磁、光或其它任何形式，在信道上传递的信息大多以二进制比特的形式表示和存在，而每一个具体的编码方案都对应于一棵二叉编码树。同一字符集的所有编码方案中，平均编码长度最小者称作最优方案。现已知该字符集中各个字符的出现频率，求最优的编码方案。	
基本要求	程序要添加适当的注释，程序的书写要采用缩进格式。 程序要具有一定的健壮性，即当输入数据非法时，程序也能适当地做出反应，如插入删除时指定的位置不对等等。 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。 根据实验报告模板详细书写实验报告，在实验报告中给出主要算法的复杂度分析。	
选做要求	无	
	已完成选做内容（序号）	无
数据结构设计	<pre> * 定义赫夫曼树的结点 */ typedef struct {     unsigned int weight; //权值     unsigned int parent, lchild, rchild; //下标 } HTNode, * HuffmanTree; /* 存储赫夫曼编码的编码表 */ typedef string* HuffmanCode; //二级指针  struct HuffChar {     char ch;     unsigned int weight; };         </pre>	

```

/*****
函数名称: HuffmanCoding
功    能: 根据权值表, 建立霍夫曼树和编码表
输入参数: 霍夫曼树根节点引用 HT, 编码表首地址引用 HC,
          权值表首地址w, 权值表长度n (支持字符数)
说    明: 思路是, 长度为n的编码表 (最多支持n种字符) w赋值给HT[1...n]
          合并n-1次, 产生n-1个新节点, 第i次合并产生的新节点下标n+i
          用静态链表方式parent, lchild, rchild记录亲子关系
*****/
void HuffmanCoding(HuffmanTree& HT, HuffmanCode& HC, HuffChar* w, int n);

/*****
函数名称: encode
功    能: 根据霍夫曼编码表将待编码的文件编码成01串
输入参数: HC: 编码表
          n: 支持字符数
          w: 权值表
          sourcefilename    : 待编码文件
          bitstreamfilename : 生成的01串文件名
*****/
void encode(HuffmanCode& HC, int n, HuffChar* w, const char const* sourcefilename,
const char const* bitstreamfilename);

/*****
函数名称: decode
功    能: 根据霍夫曼树, 将01串解码成文件
输入参数: HC: 编码表
          n: 支持字符数
          w: 权值表
          decodedfilename   : 解码文件
          bitstreamfilename : 生成的01串文件名
*****/
void decode(HuffmanTree& HT, int n, HuffChar* w, const char const*
bitstreamfilename,const char const* decodedfilename);

```

```
=====
1. 给定参考文件生成霍夫曼编码测试
2. 随机字符频率生成霍夫曼编码测试
0. 退出
=====
```

```
请选择[0/1/2]:
```

```
1
```

```
请输入编码参考文件名:test.txt
```

```
测试完毕
```

```
生成霍夫曼编码的参考文件: test.txt
```

```
参考文件的霍夫曼编码01串文件: test1_0lcode.txt
```

```
根据 test1_0lcode.txt 解码得到的文件: test1_decode.txt
```

```
comp命令比较两文件: comp test.txt test1_decode.txt
```

```
比较 test.txt 和 test1_decode.txt...
```

```
文件比较无误
```

```
是否要比较更多文件 (Y/N)? N
```

```
=====
1. 给定参考文件生成霍夫曼编码测试
2. 随机字符频率生成霍夫曼编码测试
0. 退出
=====
```

```
请选择[0/1/2]:
```

```
2
```

```
测试完毕
```

```
按照随机频率生成的随机测试文件: test2_src.txt
```

```
测试文件的霍夫曼编码01串文件: test2_0lcode.txt
```

```
根据 test2_0lcode.txt 解码得到的文件: test2_decode.txt
```

```
comp命令比较两文件: comp test2_src.txt test2_decode.txt
```

```
比较 test2_src.txt 和 test2_decode.txt...
```

```
文件比较无误
```

```
是否要比较更多文件 (Y/N)? N
```







```
=====
1. 给定参考文件生成霍夫曼编码测试
2. 随机字符频率生成霍夫曼编码测试
0. 退出
=====
```

```
请选择[0/1/2]:
```

```
0
```

提供两种测试，菜单项 1 是给定一个参考文件，根据参考文件出现的字符频率，生成霍夫曼编码，并将参考文件编码再解码，嵌入 windows 的 comp 命令，验证解码后得到的文件与原有的参考文件内容一致。菜单项 2 是随机生成字符频率表，根据字符频率表随机生成约 1MB 大

小的测试文件，对测试文件编码再解码，同样用 comp 比较测试文件和解码文件。上述 1、2 分别执行后，可以看到文件夹中有 6 个文件。

 test.txt	2020/11/20 20:04	TXT 文件	2 KB
 test1_01code.txt	2020/11/22 15:38	TXT 文件	6 KB
 test1_decode.txt	2020/11/22 15:38	TXT 文件	2 KB
 test2_01code.txt	2020/11/22 15:38	TXT 文件	4,184 KB
 test2_decode.txt	2020/11/22 15:38	TXT 文件	540 KB
 test2_src.txt	2020/11/22 15:38	TXT 文件	540 KB

程序可以抗错误输入。

```
=====
1. 给定参考文件生成霍夫曼编码测试
2. 随机字符频率生成霍夫曼编码测试
0. 退出
=====
```

请选择[0/1/2]:

```
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
输入错误, 请重新输入
```

1

请输入编码参考文件名:sadf llhjasdf iuyaj hdsgf

文件名有误, 无法打开, 请重新输入

运行结束后，程序中采用 system 函数调用 windows 的 comp 命令比较文件，发现确实无误。

对于 1 使用的测试文件 test.txt，导出其权重表、霍夫曼树和霍夫曼编码表：（树形图根据由二叉树相关函数绘制）

```
Merging [I]:0      with [~]:2      ...
Merging [^]:2      with [}:7      ...
Merging [^]:9      with [@]:11     ...
Merging [1]:11     with [N]:12     ...
Merging [□]:13     with [j]:16     ...
Merging [e]:16     with [\\]:16    ...
```

Merging [f]:17	with [!]:17	...
Merging [^]:20	with [:]20	...
Merging [^]:23	with [h]:23	...
Merging [Y]:24	with [B]:24	...
Merging [3]:24	with [(]:25	...
Merging [[]:26	with [s]:28	...
Merging [^]:29	with [7]:30	...
Merging [2]:30	with [^]:32	...
Merging [x]:32	with [u]:32	...
Merging [^]:34	with [,]:34	...
Merging [>]:35	with [<]:38	...
Merging [n]:39	with [U]:39	...
Merging [&]:39	with [^]:40	...
Merging [[]:40	with [ ]:41	...
Merging [^]:42	with [4]:42	...
Merging [0]:43	with [H]:44	...
Merging [^]:46	with [v]:46	...
Merging [8]:46	with [*]:47	...
Merging [^]:48	with [^]:49	...
Merging [^]:50	with [^]:54	...
Merging [G]:54	with [#]:55	...
Merging [^]:59	with [t]:60	...
Merging [b]:61	with [F]:61	...
Merging [^]:61	with [^]:62	...
Merging [^]:64	with [g]:66	...
Merging [W]:67	with [T]:67	...
Merging [^]:68	with [p]:72	...
Merging [I]:72	with [^]:73	...
Merging [K]:75	with [d]:76	...
Merging [q]:77	with [^]:78	...
Merging [?]:78	with [:]:78	...
Merging [^]:79	with [J]:80	...
Merging [^]:81	with [9]:82	...
Merging [[]:83	with [D]:83	...
Merging [^]:84	with [L]:84	...
Merging [^]:84	with [O]:85	...
Merging [c]:86	with [R]:86	...
Merging [P]:86	with [^]:87	...
Merging [i]:87	with [6]:89	...
Merging [{]:90	with [5]:90	...
Merging [^]:92	with [S]:92	...
Merging [=]:92	with [^]:93	...
Merging [z]:93	with [a]:94	...
Merging [^]:97	with [k]:98	...

Merging [r]:99	with [A]:99	...
Merging [-]:99	with [Z]:100	...
Merging [E]:100	with [+]:101	...
Merging [Q]:103	with [%]:103	...
Merging [^]:104	with [o]:104	...
Merging [\$]:104	with [m]:105	...
Merging [^]:109	with [y]:109	...
Merging [C]:109	with [)]:110	...
Merging [V]:111	with [X]:113	...
Merging [M]:113	with [^]:119	...
Merging [w]:119	with [ ]:120	...
Merging [.] :121	with [^]:122	...
Merging [/]:122	with [^]:123	...
Merging [^]:130	with [^]:134	...
Merging [^]:140	with [^]:145	...
Merging [^]:151	with [^]:155	...
Merging [^]:156	with [^]:159	...
Merging [^]:163	with [^]:166	...
Merging [^]:168	with [^]:169	...
Merging [^]:172	with [^]:173	...
Merging [^]:176	with [^]:180	...
Merging [^]:184	with [^]:185	...
Merging [^]:187	with [^]:195	...
Merging [^]:198	with [^]:199	...
Merging [^]:201	with [^]:206	...
Merging [^]:208	with [^]:209	...
Merging [^]:218	with [^]:219	...
Merging [^]:224	with [^]:232	...
Merging [^]:239	with [^]:243	...
Merging [^]:245	with [^]:264	...
Merging [^]:285	with [^]:306	...
Merging [^]:315	with [^]:329	...
Merging [^]:337	with [^]:345	...
Merging [^]:356	with [^]:369	...
Merging [^]:382	with [^]:397	...
Merging [^]:407	with [^]:417	...
Merging [^]:437	with [^]:456	...
Merging [^]:482	with [^]:509	...
Merging [^]:591	with [^]:644	...
Merging [^]:682	with [^]:725	...
Merging [^]:779	with [^]:824	...
Merging [^]:893	with [^]:991	...
Merging [^]:1235	with [^]:1407	...
Merging [^]:1603	with [^]:1884	...

Merging [^]:2642 with [^]:3487 ...

[T]:67	*			┌─[T]:67	1111111
[^]:134	*			┌─[^]:134	
[W]:67	*			┌─[W]:67	1111110
[^]:264	*			┌─[^]:264	
[g]:66	*			┌─[g]:66	1111101
[^]:130	*			┌─[^]:130	
[u]:32	*			┌─[u]:32	11111001
[^]:64	*			┌─[^]:64	
[x]:32	*			┌─[x]:32	11111000
[^]:509	*			┌─[^]:509	
[\\]:16	*			┌─[\\]:16	111101111
[^]:32	*			┌─[^]:32	
[e]:16	*			┌─[e]:16	111101110
[^]:62	*			┌─[^]:62	
[2]:30	*			┌─[2]:30	11110110
[^]:123	*			┌─[^]:123	
["]:61	*			┌─["]:61	1111010
[^]:245	*			┌─[^]:245	
[/]:122	*			┌─[/]:122	111100
[^]:991	*			┌─[^]:991	
[F]:61	*			┌─[F]:61	1110111
[^]:122	*			┌─[^]:122	
[b]:61	*			┌─[b]:61	1110110
[^]:243	*			┌─[^]:243	
[.]:121	*			┌─[.]:121	111010
[^]:482	*			┌─[^]:482	
[_]:120	*			┌─[_]:120	111001
[^]:239	*			┌─[^]:239	
[w]:119	*			┌─[w]:119	111000
[^]:1884	*			┌─[^]:1884	
[t]:60	*			┌─[t]:60	1101111
[^]:119	*			┌─[^]:119	
[7]:30	*			┌─[7]:30	11011101
[^]:59	*			┌─[^]:59	
[j]:16	*			┌─[j]:16	110111001
[^]:29	*			┌─[^]:29	
[@]:13	*			┌─[@]:13	110111000
[^]:232	*			┌─[^]:232	
[M]:113	*			┌─[M]:113	110110
[^]:456	*			┌─[^]:456	
[X]:113	*			┌─[X]:113	110101
[^]:224	*			┌─[^]:224	

[V]:111	*					└─[V]:111	110100
[^]:893	*				└─[^]:893		
[]:110	*					└─[]:110	110011
[^]:219	*					└─[^]:219	
[C]:109	*					└─[C]:109	110010
[^]:437	*					└─[^]:437	
[y]:109	*					└─[y]:109	110001
[^]:218	*					└─[^]:218	
[#]:55	*					└─[#]:55	1100001
[^]:109	*					└─[^]:109	
[G]:54	*					└─[G]:54	1100000
[^]:3487	*		└─[^]:3487				
[m]:105	*					└─[m]:105	101111
[^]:209	*					└─[^]:209	
[\$]:104	*					└─[\$]:104	101110
[^]:417	*					└─[^]:417	
[o]:104	*					└─[o]:104	101101
[^]:208	*					└─[^]:208	
[s]:28	*					└─[s]:28	10110011
[^]:54	*					└─[^]:54	
[ ]:26	*					└─[ ]:26	10110010
[^]:104	*					└─[^]:104	
[^]:50	*					└─[^]:50	1011000
[^]:824	*			└─[^]:824			
[%]:103	*					└─[%]:103	101011
[^]:206	*					└─[^]:206	
[Q]:103	*					└─[Q]:103	101010
[^]:407	*					└─[^]:407	
[+]:101	*					└─[+]:101	101001
[^]:201	*					└─[^]:201	
[E]:100	*					└─[E]:100	101000
[^]:1603	*			└─[^]:1603			
[Z]:100	*					└─[Z]:100	100111
[^]:199	*					└─[^]:199	
[-]:99	*					└─[-]:99	100110
[^]:397	*					└─[^]:397	
[A]:99	*					└─[A]:99	100101
[^]:198	*					└─[^]:198	
[r]:99	*					└─[r]:99	100100
[^]:779	*			└─[^]:779			
[k]:98	*					└─[k]:98	100011
[^]:195	*					└─[^]:195	
[():25	*					└─[():25	10001011
[^]:49	*					└─[^]:49	



[3]:24	*							└─[3]:24	10001010
[^]:97	*						└─[^]:97		
[B]:24	*						└─[B]:24	10001001	
[^]:48	*						└─[^]:48		
[Y]:24	*						└─[Y]:24	10001000	
[^]:382	*						└─[^]:382		
[a]:94	*						└─[a]:94	100001	
[^]:187	*						└─[^]:187		
[z]:93	*						└─[z]:93	100000	
[^]:6129	*—	[^]:6129							
[*]:47	*						└─[*]:47	0111111	
[^]:93	*						└─[^]:93		
[8]:46	*						└─[8]:46	0111110	
[^]:185	*						└─[^]:185		
[=]:92	*						└─[=]:92	011110	
[^]:369	*						└─[^]:369		
[S]:92	*						└─[S]:92	011101	
[^]:184	*						└─[^]:184		
[v]:46	*						└─[v]:46	0111001	
[^]:92	*						└─[^]:92		
[h]:23	*						└─[h]:23	01110001	
[^]:46	*						└─[^]:46		
[N]:12	*						└─[N]:12	011100001	
[^]:23	*						└─[^]:23		
[1]:11	*						└─[1]:11	011100000	
[^]:725	*						└─[^]:725		
[5]:90	*						└─[5]:90	011011	
[^]:180	*						└─[^]:180		
[{}]:90	*						└─[{}]:90	011010	
[^]:356	*						└─[^]:356		
[6]:89	*						└─[6]:89	011001	
[^]:176	*						└─[^]:176		
[i]:87	*						└─[i]:87	011000	
[^]:1407	*						└─[^]:1407		
[H]:44	*						└─[H]:44	0101111	
[^]:87	*						└─[^]:87		
[0]:43	*						└─[0]:43	0101110	
[^]:173	*						└─[^]:173		
[P]:86	*						└─[P]:86	010110	
[^]:345	*						└─[^]:345		
[R]:86	*						└─[R]:86	010101	
[^]:172	*						└─[^]:172		
[c]:86	*						└─[c]:86	010100	
[^]:682	*						└─[^]:682		

[O]:85	*				┌[O]:85	010011	
[^]:169	*				┌[^]:169		
[']:84	*				└[']:84	010010	
[^]:337	*			└[^]:337			
[L]:84	*				┌[L]:84	010001	
[^]:168	*			└[^]:168			
[4]:42	*				┌[4]:42	0100001	
[^]:84	*			└[^]:84			
[`]:42	*			└[`]:42	0100000		
[^]:2642	*	└[^]:2642					
[D]:83	*			┌[D]:83	001111		
[^]:166	*			┌[^]:166			
[ ]:83	*			└[ ]:83	001110		
[^]:329	*			┌[^]:329			
[9]:82	*			┌[9]:82	001101		
[^]:163	*			└[^]:163			
[ ]:41	*				┌[ ]:41	0011001	
[^]:81	*			└[^]:81			
[[]:40	*			└ [[]:40	0011000		
[^]:644	*		┌[^]:644				
[J]:80	*				┌[J]:80	001011	
[^]:159	*				┌[^]:159		
[,]:20	*					┌[,]:20 00101011	
[^]:40	*					┌[^]:40	
[@]:11	*					┌[@]:11 001010101	
[^]:20	*					└[^]:20	
[ ]:7	*					┌[ ]:7 0010101001	
[^]:9	*					└[^]:9	
[~]:2	*						┌[~]:2 00101010001
[^]:2	*					└[^]:2	
[l]:0	*					└[l]:0	00101010000
[^]:79	*				└[^]:79		
[&]:39	*				└[&]:39	0010100	
[^]:315	*			└[^]:315			
[,]:78	*				┌[,]:78	001001	
[^]:156	*			└[^]:156			
[?]:78	*			└[?]:78	001000		
[^]:1235	*	└[^]:1235					
[U]:39	*			┌[U]:39	0001111		
[^]:78	*			┌[^]:78			
[n]:39	*				└[n]:39	0001110	
[^]:155	*			┌[^]:155			
[q]:77	*			└[q]:77	000110		
[^]:306	*		┌[^]:306				

[d]:76	*				┐[d]:76	000101
[^]:151	*			└[^]:151		
[K]:75	*			└[K]:75	000100	
[^]:591	*	└[^]:591				
[<]:38	*			┐[<]:38	0000111	
[^]:73	*			┐[^]:73		
[>]:35	*			└[>]:35	0000110	
[^]:145	*			┐[^]:145		
[l]:72	*			└[l]:72	000010	
[^]:285	*	└[^]:285				
[p]:72	*			┐[p]:72	000001	
[^]:140	*	└[^]:140				
[,]:34	*			┐[,]:34	0000001	
[^]:68	*	└[^]:68				
[!]:17	*			┐[!]:17	00000001	
[^]:34	*	└[^]:34				
[f]:17	*	└[f]:17			00000000	

: 0011001

!: 00000001

": 1111010

#: 1100001

\$.: 101110

%.: 101011

&.: 0010100

': 010010

(: 10001011

): 110011

\*.: 0111111

+.: 101001

,.: 0000001

-.: 100110

..: 111010

/.: 111100

0.: 0101110

1.: 011100000

2.: 11110110

3.: 10001010

4.: 0100001

5.: 011011

6.: 011001

7.: 11011101

8.: 0111110

9.: 001101

	:: 001001
	;: 00101011
	<: 0000111
	=: 011110
	>: 0000110
	?: 001000
	@: 001010101
	A: 100101
	B: 10001001
	C: 110010
	D: 001111
	E: 101000
	F: 1110111
	G: 1100000
	H: 0101111
	I: 000010
	J: 001011
	K: 000100
	L: 010001
	M: 110110
	N: 011100001
	O: 010011
	P: 010110
	Q: 101010
	R: 010101
	S: 011101
	T: 1111111
	U: 0001111
	V: 110100
	W: 1111110
	X: 110101
	Y: 10001000
	Z: 100111
	[: 0011000
	\: 111101111
	]: 001110
	^: 1011000
	_ : 111001
	`: 0100000
	a: 100001
	b: 1110110
	c: 010100
	d: 000101
	e: 111101110


f: 00000000  
g: 1111101  
h: 01110001  
i: 011000  
j: 110111001  
k: 100011  
l: 00101010000  
m: 101111  
n: 0001110  
o: 101101  
p: 000001  
q: 000110  
r: 100100  
s: 10110011  
t: 1101111  
u: 11111001  
v: 0111001  
w: 111000  
x: 11111000  
y: 110001  
z: 100000  
{: 011010  
|: 10110010  
}: 0010101001  
~: 00101010001  
□: 110111000

源文件 test.txt 共 1043 字节。



test.txt

文件类型: TXT 文件 (.txt)



打开方式:  Notepad++ : a free (G

更改(C)...

位置: D:\homework\WY\WYDS5\5-5

大小: 1.01 KB (1,043 字节)

导出的编码文件 test1\_01code.txt 文件为 5924 字节。实际 5924 个 0/1 传输需要 5924bit，压缩率  $5924/(1043*8) = 71\%$ 。

	<div><div>test1_01code.txt</div><div>文件类型: TXT 文件 (.txt)</div><div>打开方式:  Notepad++ : a free (G <span>更改(C)...</span></div><div>位置: D:\homework\WY\WYDS5\5-5</div><div>大小: 5.78 KB (5,924 字节)</div></div>
心得体会	<p>生成霍夫曼树和编码表的函数 HuffmanCoding，每次查找当前没有父亲节点的权值最小的两个节点的函数为 Select。Select 线性实现 <math>O(n)</math>，二分查找 <math>O(\log n)</math>。</p> <p>我实现的 Select 函数复杂度为 <math>O(n)</math>，HuffmanCoding 函数复杂度为 <math>O(n^2)</math>。若采用左式堆/优先队列等方法，Select 函数复杂度可以降至 <math>O(\log n)</math>，HuffmanCoding 函数复杂度降至 <math>O(n \log n)</math>。现实中，<math>n \leq 256</math>，可以认为，在已知字符频率的条件下，生成霍夫曼编码的时间是常数级。</p> <p>编码函数 encode，复杂度 <math>O(n)</math>，<math>n</math> 为要编码的文件大小。由于已经建立了编码表，可以 <math>O(1)</math> 直接取得读取到的字符的编码，因此复杂度降为 <math>O(n)</math>。</p> <p>译码函数 decode，复杂度 <math>O(n)</math>，<math>n</math> 为 0/1bit 位数。每次读取一个 0/1，决定向左子树还是右子树移动一步。若当前移动后到达叶子节点，解码出一个字符，返回霍夫曼树树根。</p> <p>霍夫曼编码的核心算法思想是贪心。要使得整体的编码长度最短，应该贪心地让出现次数越多的字符编码越短。0/1 二进制与二叉树结构有着天然联系。</p>
代码实现	<pre>#define _CRT_SECURE_NO_WARNINGS #include &lt;iostream&gt; #include &lt;fstream&gt; #include &lt;cstdio&gt; #include &lt;cstdlib&gt; #include &lt;iomanip&gt; #include &lt;string&gt; #include &lt;conio.h&gt; using namespace std;  #define MB 1024*1024  #ifdef _MSC_VER #pragma warning(disable:4114) #endif</pre>

```

/* 定义赫夫曼树的结点 */
typedef struct {
    unsigned int weight; //权值
    unsigned int parent, lchild, rchild; //下标
} HTNode, * HuffmanTree;
/* 存储赫夫曼编码的编码表 */
typedef string* HuffmanCode; //二级指针

struct HuffChar
{
    char ch;
    unsigned int weight;
};

/*****
函数名称: Select
功    能: 从HT[1...k]中挑选出没有parent的节点中权值最小的两个
          下标分别引用传给s1, s2
输入参数: 霍夫曼树首地址HT, k, s1, s2
*****/
void Select(const HuffmanTree& HT, const int k, int& s1, int& s2)
{
    if (!HT)
        return;
    HTNode t1;
    HTNode t2;
    t1.weight = t2.weight = 0x7fffffff; //初始化
    int ss1 = 0;
    int ss2 = 0;
    for (int i = 1; i <= k; i++)
    {
        if (!HT[i].parent)
        {
            if (HT[i].weight <= t1.weight)
            {
                t2 = t1;
                t1 = HT[i];
                ss2 = ss1;
                ss1 = i;
            }
            else if (HT[i].weight < t2.weight)
            {
                t2 = HT[i];
            }
        }
    }
}

```

```

        ss2 = i;
    }
}
}
if(ss1)//找到才赋值
    s1 = ss1;
if(ss2)
    s2 = ss2;
}

/*****
函数名称: HuffmanCoding
功    能: 根据权值表, 建立霍夫曼树和编码表
输入参数: 霍夫曼树根节点引用 HT, 编码表首地址引用 HC,
          权值表首地址w, 权值表长度n (支持字符数)
说    明: 思路是, 长度为n的编码表 (最多支持n种字符) w赋值给HT[1...n]
          合并n-1次, 产生n-1个新节点, 第i次合并产生的新节点下标n+i
          用静态链表方式parent, lchild, rchild记录亲子关系
*****/
void HuffmanCoding(HuffmanTree& HT, HuffmanCode& HC, HuffChar* w, int n)
{
    //HT : 赫夫曼树
    //HC : 赫夫曼编码
    //w : int型数组, 存放n的字符的出现频率表, { 7,5,2,3, ...}形式
    //n : 字符的数量
    if (!w)
        return;

    int i, m, s1, s2, c, f;
    string cd;
    if (n <= 1) //仅一个结点无法构造赫夫曼树
        return;
    m = 2 * n - 1; //n是字符数, 即叶子结点数n0, n0=n2+1, 无n1, 总数=2n-1

    HT = new(nothrow) HTNode[m + 1]; // [0]不用
    if (!HT)
        return;

    /* w[0~(n-1)] 赋值给 HT[1~n] */
    for (i = 1; i <= n; i++)
    {
        HT[i].weight = w[i-1].weight;
        HT[i].parent = 0;
        HT[i].lchild = 0;
    }
}

```



```

    HT[i].rchild = 0;
}

/* 初始化所有后续加入的枝干节点 */
for (; i <= m; i++)
{
    HT[i].weight = 0;
    HT[i].parent = 0;
    HT[i].lchild = 0;
    HT[i].rchild = 0;
}
/* n+1到m都是用做非叶结点的，依次使用即可 */
for (i = n + 1; i <= m; i++)
{
    /* Select 是在HT[1..i-1]中选择parent为0条件下的权值最小结点
    返回的下标在s1, s2中 (需要单独给出其实现) */
    Select(HT, i - 1, s1, s2);
    /* 第i个结点称为s1,s2的父结点，权值为两者之和
    s1/s2的parent被置为i，下次不会被Select选中 */

    HT[s1].parent = i;
    HT[s2].parent = i;
    //s1与s2的父亲是新加入的i

    HT[i].lchild = s1;
    HT[i].rchild = s2;
    //i的左右孩子是s1和s2

    HT[i].weight = HT[s1].weight + HT[s2].weight;
    //合并后节点权值=左右孩子权值之和

}

HC = new(nothrow) string[n + 1]; //n+1, [0]不用
if (!HC)
    return;

for (i = 1; i <= n; i++)
{ //对前n个结点（叶子结点）循环
    string cd = ""; //临时字符串
    for (c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent)
        if (HT[f].lchild == c) //判断左右子树分取01
            cd = string("0") + cd;
        else

```

```

        cd = string("1") + cd;
        HC[i] = cd;
    }
}

/* 二分查找c在数组w中的下标位置，w数组没排过序，就是字典序 */
int index(HuffChar* w, int n, const char c)
{
    int l = 0;
    int r = n;
    while (l < r)
    {
        int mid = l + (r - l) / 2;
        if (unsigned char(w[mid].ch) < unsigned char(c))
            l = mid + 1;
        else
            r = mid;
    }
    return l + 1; //返回下标+1,因为霍夫曼树[0]节点不使用
}

*****
函数名称: encode
功    能: 根据霍夫曼编码表将待编码的文件编码成01串
输入参数: HC: 编码表
           n: 支持字符数
           w: 权值表
           sourcefilename    : 待编码文件
           bitstreamfilename : 生成的01串文件名
*****/

void encode(HuffmanCode& HC, int n, HuffChar* w, const char const* sourcefilename,
const char const* bitstreamfilename)
{
    ifstream fin(sourcefilename, ios::in | ios::binary);
    if (!fin.is_open())
        return;
    ofstream fout(bitstreamfilename, ios::out | ios::binary);
    if (!fout.is_open())
        return;
    char c;
    while (1)
    {
        fin.get(c);
        if (fin.eof())

```

```

        break;
        int ind = index(w,n,c);
        fout << HC[ind];
    }
    fin.close();
    fout.close();
}

/*****
函数名称: decode
功    能: 根据霍夫曼树, 将01串解码成文件
输入参数: HC: 编码表
           n: 支持字符数
           w: 权值表
           decodedfilename : 解码文件
           bitstreamfilename : 生成的01串文件名
*****/
void decode(HuffmanTree& HT, int n, HuffChar* w, const char const*
bitstreamfilename,const char const* decodedfilename)
{
    int m = 2 * n - 1;
    int p = m;

    ifstream fin(bitstreamfilename, ios::in | ios::binary);
    if (!fin.is_open())
        return;

    ofstream fout(decodedfilename, ios::out | ios::binary);
    if (!fout.is_open())
        return;

    char ch;
    while (1)
    {
        fin.get(ch);
        if (fin.eof())
            break;
        if (ch == '0')//遇0向左子树走
            p = HT[p].lchild;
        if (ch == '1')//遇1向右子树走
            p = HT[p].rchild;
        if (!HT[p].lchild && !HT[p].rchild)//走到叶子
        {
            fout << w[p - 1].ch;//注意下标对应关系
        }
    }
}

```

```

        p = m;
    }
}
fin.close();
fout.close();
}

/*****
函数名称: statis
功    能: 根据参考文件中各个字符出现频率生成权重表
输入参数: n: 支持字符数
          sourcefilename    : 权值频率参考文件
返 回 值: 权值表动态数组首地址
*****/
HuffChar* statis(const char const* sourcefilename,int &n)
{
    HuffChar* w1 = new(nothrow) HuffChar[256];//开到最大
    if (!w1)
        return NULL;

    for (int i = 0; i < 256; i++)
    {
        w1[i].ch = char(i);
        w1[i].weight = 0;
    }

    ifstream fin(sourcefilename, ios::in | ios::binary);
    if (!fin.is_open())
        return NULL;

    char c;
    while (1)
    {
        fin.get(c);
        if (fin.eof())
            break;
        if (!w1[unsigned char(c)].weight)//该字符在文件中第一次出现, 种类数++
            n++;
        w1[unsigned char(c)].weight++;//该字符权重/频率++
    }

    HuffChar* w2 = new(nothrow) HuffChar[n];
    if (!w2)
        return NULL;
}

```

```

int j = 0;
for (int i = 0; i < 256; i++)
    if (w1[i].weight > 0)//找出所有出现过的，放进新表w2
        w2[j++] = w1[i];

delete[]w1;//释放原表w1

fin.close();

return w2;
}

/*****
函数名称: maintest
功    能: 给定参考文件生成霍夫曼编码测试
输入参数: sourcefilename    : 权值频率参考文件
*****/
int maintest(const char const* sourcefilename)
{
    int n = 0;
    HuffmanTree HT;
    HuffmanCode HC;

    HuffChar* w = statis(sourcefilename, n);
    if (!w)
        return 0;

    HuffmanCoding(HT, HC, w, n);
    encode(HC, n, w, sourcefilename, "test1_01code.txt");
    decode(HT, n, w, "test1_01code.txt", "test1_decode.txt");

    cout << "测试完毕" << endl;
    cout << "生成霍夫曼编码的参考文件: " << sourcefilename << endl;
    cout << "参考文件的霍夫曼编码01串文件: test1_01code.txt" << endl;
    cout << "根据 test1_01code.txt 解码得到的文件: test1_decode.txt" << endl;
    cout << "comp命令比较两文件: comp " << sourcefilename << " test1_decode.txt"
<< endl;
    string command = string("comp ") + string(sourcefilename) + string("
test1_decode.txt");
    system(command.c_str());

    delete[]HT;
    delete[]HC;
}

```

```
delete[]w;
```

```
return 0;
```

```
}
```

```
/******
```

函数名称: **randtest**

功 能: 随机字符频率生成霍夫曼编码测试

说 明: 随机产生包含**ASCII = 0~254 (EOF除外)**的**255**个字符的权重频率表

(保证每个都有)随机产生与该表相符的文件, 大小期望**1MB**

生成该随机测试文件的霍夫曼编码**01**串, 再解码一次

测试比较随机源文件和解码文件

```
*****/
```

```
int randtest()
```

```
{
```

```
int n = 255;
```

```
HuffmanTree HT;
```

```
HuffmanCode HC;
```

```
HuffChar* w = new(nothrow) HuffChar[255];
```

```
if (!w)
```

```
return 0;
```

```
int tot = 0;
```

```
int count[255];
```

```
for (int i = 0; i < n; i++)
```

```
{
```

```
int c = rand() % (MB / 256) + 1;
```

```
tot += c;
```

```
w[i].weight = c;
```

```
w[i].ch = char(i);
```

```
count[i] = c;
```

```
}
```

```
ofstream fout("test2_src.txt", ios::out | ios::binary);
```

```
if (!fout.is_open())
```

```
return 0;
```

```
while (tot > 0)
```

```
{
```

```
int i = rand() % 255;
```

```
if (count[i] > 0)
```

```
{
```

```

        tot--;
        count[i]--;
        fout << char(i);
    }
}

fout.close();

HuffmanCoding(HT, HC, w, n);
encode(HC, n, w, "test2_src.txt", "test2_01code.txt");
decode(HT, n, w, "test2_01code.txt", "test2_decode.txt");

cout << "测试完毕" << endl;
cout << "按照随机频率生成的随机测试文件: test2_src.txt" << endl;
cout << "测试文件的霍夫曼编码01串文件: test2_01code.txt" << endl;
cout << "根据 test2_01code.txt 解码得到的文件: test2_decode.txt" << endl;
cout << "comp命令比较两文件: comp test2_src.txt test2_decode.txt" << endl;
system("comp test2_src.txt test2_decode.txt");

delete[]HT;
delete[]HC;
delete[]w;

return 0;
}

int usage()//菜单函数
{
    cout << "===== " << endl;
    cout << "1.给定参考文件生成霍夫曼编码测试" << endl;
    cout << "2.随机字符频率生成霍夫曼编码测试" << endl;
    cout << "0.退出" << endl;
    cout << "===== " << endl;
    cout << "请选择[0/1/2]: " << endl;
    while (1)
    {
        char ch = _getch();
        if (ch == '1')
        {
            cout << "1" << endl;
            return 1;
        }
        if (ch == '2')

```

```

    {
        cout << "2" << endl;
        return 2;
    }
    if (ch == '0')
    {
        cout << "0" << endl;
        return 0;
    }
    cout << "输入错误，请重新输入" << endl;
}
}

int main()
{
    while (1)
    {
        int sel = usage();
        if (!sel) break;
        if (sel == 1)
        {
            string sourcefilename;
            while (1)
            {
                cout << "请输入编码参考文件名:";
                cin >> sourcefilename;
                ifstream fin(sourcefilename.c_str(), ios::in | ios::binary);
                if (!fin.is_open())
                {
                    cerr << "文件名有误，无法打开，请重新输入" << endl;
                }
                else
                {
                    fin.close();
                    break;
                }
            }
            maintest(sourcefilename.c_str());
        }
        else
        {
            randtest();
        }
    }
}

```



```
}  
    return 0;
```