

《数据结构》上机实验报告

第 6 次上机

学号：

姓名：

学院： 信息科学学院

专业： 计算机科学

教师：

日期： 2018.11.5

1. 实验要求

1.上机之前应做好充分准备，认真思考所需的上机题目，提高上机效率。

2.独立上机输入和调试自己所编的程序，切忌抄袭、拷贝他人程序。

3.上机结束后，整理出实验报告。书写报告时，重点放在实验的方法、思路以及总结反思上，以达到巩固课堂学习、提高动手能力的目的。

1. 实验内容

用最小堆编程实现给定权值集合下构造相应霍夫曼树的算法，并解决以下问题：

有一电文共使用五种字符a,b,c,d,e,其出现频率依次为4,7,5,2,9。

(1)构造对应的编码哈夫曼树(要求左子树根结点的权小于等于右子树根结点的权)。

(2)给出每个字符的哈夫曼编码。

(3)译出编码系列11000111000101011的相应电文。

三、实验步骤（写出问题分析或者算法思路）

1. WPL是树的所有叶结点的带权路径长度之和，称为树的带权路径长度。把WPL最小的树称为哈夫曼树。这个题中默认把叶结点到根的路径长度当作权值，即叶结点的深度减一，那么简单的想法就是把叶节点中较大值放在深度小的地方，叶节点中较小值放在深度大的地方。
2. 如果只有一个叶节点，那么它自己成为根结点，此时它的权值是0。如果只有两个结点，那么一个作为左结点，一个作为右结点就可以了，此时俩结点的权值都是1。
3. 哈夫曼树的每个结点的度必须是0（叶节点）或者2，因为总是可以把度为1的结点给删去，从而得到一个WPL更小的树。换句话说哈夫曼树上的非根结点总是有它的兄弟结点。
4. 一个哈夫曼树中，叶节点值小的深度一定比叶节点值大的深度大。否则的话互换大小结点的位置就能得到一个WPL更小的新树。
5. 考虑如何从一个哈夫曼树推出另一个哈夫曼树：

用A表示叶节点集合，用表示以集合A中的点作为叶节点构造出来的哈夫曼树的带权路径长度，用表示叶节点的值，表示叶节点的深度。。

由树深度的定义，至少有一个点的深度恰好是树的深度，由第3点知道，这个特殊点一定有一个兄弟结点，由第4点知道这两个点是和。如果把它们俩删除，他俩的父节点构成一个新的叶结点，的值是

得到的新集合是

那么这棵新树的WPL是：

下面证明新集合的哈夫曼树就是新树，即证明：

若有一个树以新集合为叶节点，并且它的WPL 小于的话，那么把它中的结点分解成和后，得到的树的WPL

新的树的WPL比A的哈夫曼树的WPL还要小，矛盾。所以新集合的哈夫曼树就是新树。

1. 哈夫曼树的构造：先随机生成一个结构树，保证结构树的每个叶节点都是给定的叶节点。

1.找出两个最小的叶结点，计算出两个权值之和，然后建立真正的关系表（做好记录）

2.砍掉两个最小的叶节点，其父节点作为成员融入新树，更新父节点的权值和位置。

重复以上两点直到结构树只有一个根结点，那么建立的关系表就是最后需要的哈夫曼树。

在最终的算法中，是使用最小堆来实现的结构树。因为每次都是在寻找最小的结点，恰好最小堆专门是找最小结点。

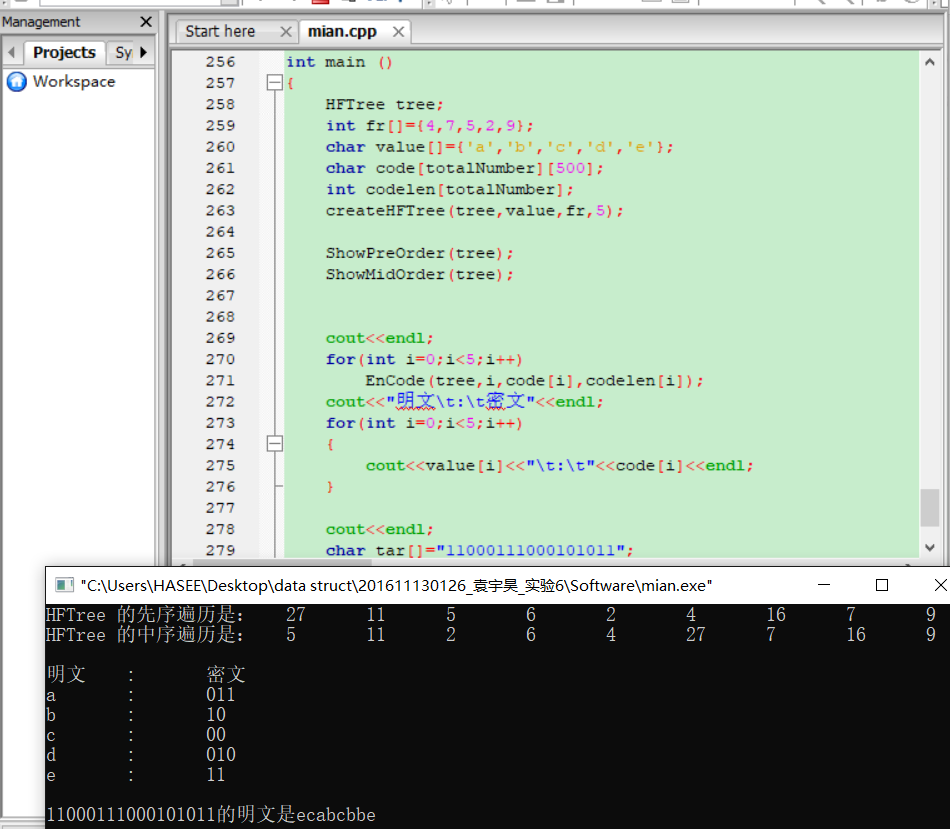
1. 字符的加密，字符密文中的1和0是由根结点到密文位置路径所决定的，所以得先找出路径，然后再加密。详见EnCode()函数。
2. 密文的解密，密文中的1和0是由根结点到密文位置路径所决定的，所以得把密文当作路径，在树中寻找明文。详见DeCode()函数。
3. 程序清单（源程序代码等）

main.cpp

1. #include<iostream>
2. **using** **namespace** std;
3. **const** **int** heapSize=1000,totalNumber=100;
5. **typedef** **struct** {
6. **char** data;             //结点的值
7. **int** weight;            //结点的权
8. **int** parent, lchild, rchild;    //双亲、左、右子女
9. } HTNode;
11. **typedef** **struct** {
12. HTNode elem[totalNumber];   //树存储数组
13. **int** num, root;             //外结点数与根
14. } HFTree;
16. **typedef** **int** HElemType;        //堆中元素的数据类型
17. **typedef** **struct** {              //堆结构的定义
18. HElemType elem[heapSize];   //小根堆存储数组
19. **int** curSize;            //当前元素个数
20. } minHeap;
22. **void** CreateMinHeap(HFTree &tree,minHeap &heap,**int** n);
23. **void** ShiftUp(HFTree &tree,minHeap &heap,**int** x);
24. **void** ShiftDown(HFTree &tree,minHeap &heap,**int** x);
25. **void** Switch(minHeap &heap,**int** a,**int** b);
26. **void** ShowHeap(HFTree &tree,minHeap &heap);
27. **void** AddHeap(HFTree &tree,minHeap &heap,**int** pos);
28. **void** DeleteHeap(HFTree & tree,minHeap &heap ,**int** &pos,**int** &minn);
29. **void** AddTree(HFTree &tree,minHeap &heap,**int** i,**int** s1,**int** min1,**int** s2,**int** min2);
30. **void** ShowPreOrder(**const** HFTree &tree);
31. **void** PreOrder(**const** **int** a,**const** HFTree &tree);
32. **void** ShowMidOrder(**const** HFTree &tree);
33. **void** MidOrder(**const** **int** a,**const** HFTree &tree);
34. **void** EnCode(**const** HFTree &tree,**int** pos,**char** code[],**int** &len);
35. **void** DeCode(**char** value[],**char** tar[],**const** HFTree &tree,**char** result[]);
37. **void** createHFTree(HFTree& tree,**char** value[],**int** fr[], **int** n)
38. {
39. **for**(**int** i=0;i<n;i++)
40. {
41. tree.elem[i].data=value[i];
42. tree.elem[i].weight=fr[i];
43. tree.elem[i].lchild=tree.elem[i].rchild=tree.elem[i].parent=-1;
44. }
45. minHeap heap;
46. CreateMinHeap(tree,heap,n);
47. //ShowHeap(tree,heap);
48. **int** s1,s2,min1,min2;
49. **for**(**int** i=n;i<2\*n-1;i++)
50. {
51. DeleteHeap(tree,heap,s1,min1);
52. DeleteHeap(tree,heap,s2,min2);
53. //cout<<"s:"<<s1<<" "<<s2<<"  min:"<<min1<<" "<<min2<<endl;
54. //ShowHeap(tree,heap);
55. AddTree(tree,heap,i,s1,min1,s2,min2);
56. //ShowHeap(tree,heap);
57. }
58. tree.num=n;
59. tree.root=2\*n-2;
60. }
62. **void** AddTree(HFTree &tree,minHeap &heap,**int** i,**int** s1,**int** min1,**int** s2,**int** min2)
63. {
64. tree.elem[s1].parent = tree.elem[s2].parent = i;
65. tree.elem[i].lchild = s1;  tree.elem[i].rchild = s2;
66. tree.elem[i].weight =min1+min2,tree.elem[i].parent=-1;
67. AddHeap(tree,heap,i);
68. }
70. **void** CreateMinHeap(HFTree &tree,minHeap &heap,**int** n)
71. {
72. **for**(**int** i=1;i<=n;i++)
73. {
74. heap.elem[i]=i-1;
75. }
76. heap.curSize=n;
77. //ShowHeap(tree,heap);
78. **for**(**int** i=heap.curSize;i>=1;i--)
79. {
80. ShiftUp(tree,heap,i);
81. }
82. }
84. **void** ShiftUp(HFTree &tree,minHeap &heap,**int** x)
85. {
86. **if**(x<=1)
87. **return** ;
88. **int** nxt=heap.elem[x/2],now=heap.elem[x];
89. //cout<<"ShiftUP"<<x<<"->"<<tree.elem[now].weight<<endl;
90. **if**(tree.elem[nxt].weight>tree.elem[now].weight)
91. {
92. //cout<<"00ShiftUP"<<x<<"->"<<tree.elem[now].weight<<endl;
93. Switch(heap,x/2,x);
94. ShiftDown(tree,heap,x);
95. ShiftUp(tree,heap,x/2);
96. }
97. }
99. **void** ShiftDown(HFTree &tree,minHeap &heap,**int** x)
100. {
101. **if**(x\*2>heap.curSize)
102. **return** ;
103. **if**(x\*2==heap.curSize)
104. {
105. **int** nxt=heap.elem[x\*2],now=heap.elem[x];
106. **if**(tree.elem[now].weight>tree.elem[nxt].weight)
107. {
108. Switch(heap,x,x\*2);
109. //ShiftDown(tree,heap,x\*2);
110. }
111. }
112. **if**(x\*2<heap.curSize)
113. {
114. **int** to\_switch=-1;
115. **int** nxt=heap.elem[x\*2],nxt2=heap.elem[x\*2+1],now=heap.elem[x];
116. **if**(tree.elem[now].weight<tree.elem[nxt].weight && tree.elem[now].weight<tree.elem[nxt2].weight)
117. **return** ;
118. **if**(tree.elem[now].weight>tree.elem[nxt].weight)
119. to\_switch=x\*2;
120. **if**(tree.elem[now].weight>tree.elem[nxt2].weight && tree.elem[nxt2].weight<tree.elem[nxt].weight)
121. to\_switch=x\*2+1;
122. Switch(heap,x,to\_switch);
123. }
124. // cout<<"Shift Down"<<nxt<<" "<<x<<endl;
125. }
126. **void** Switch(minHeap &heap,**int** a,**int** b)
127. {
128. **int** tmp=heap.elem[a];
129. heap.elem[a]=heap.elem[b];
130. heap.elem[b]=tmp;
131. **return** ;
132. }
134. **void** ShowHeap(HFTree &tree,minHeap &heap)
135. {
136. **int** cnt=1;
137. cout<<"heap size:"<<heap.curSize<<endl;
138. **for**(**int** i=1;cnt<=heap.curSize;i++)
139. {
140. **for**(**int** j=1;j<=i && cnt<=heap.curSize;j++)
141. {
142. cout<<tree.elem[heap.elem[cnt++]].weight<<" ";
143. }
144. cout<<endl;
145. }
146. }
148. **void** AddHeap(HFTree &tree,minHeap &heap,**int** pos)
149. {
150. heap.elem[++heap.curSize]=pos;
151. ShiftUp(tree,heap,heap.curSize);
152. **return**;
153. }
155. **void** DeleteHeap(HFTree & tree,minHeap &heap ,**int** &pos,**int** &minn)
156. {
157. //ShowHeap(tree,heap);
158. //cout<<"DeletHeap:"<<heap.curSize<<endl;
159. **if**(heap.curSize<=0)
160. {
161. pos=minn=-1;
162. **return** ;
163. }
164. pos=heap.elem[1];
165. minn=tree.elem[pos].weight;
166. heap.elem[1]=heap.elem[heap.curSize];
167. //ShowHeap(tree,heap);
168. heap.curSize--;
169. ShiftDown(tree,heap,1);
170. //ShowHeap(tree,heap);
171. //cout<<endl;
172. **return** ;
173. }
175. **void** ShowPreOrder(**const** HFTree &tree)
176. {
177. cout<<"HFTree 的先序遍历是：";
178. PreOrder(tree.root,tree);
179. cout<<endl;
180. }
182. **void** PreOrder(**const** **int** a,**const** HFTree &tree)
183. {
184. cout<<"\t"<<tree.elem[a].weight;
185. **if**(tree.elem[a].lchild>=0)
186. PreOrder(tree.elem[a].lchild,tree);
187. **if**(tree.elem[a].rchild>=0)
188. PreOrder(tree.elem[a].rchild,tree);
189. }
191. **void** ShowMidOrder(**const** HFTree &tree)
192. {
193. cout<<"HFTree 的中序遍历是：";
194. MidOrder(tree.root,tree);
195. cout<<endl;
196. }
198. **void** MidOrder(**const** **int** a,**const** HFTree &tree)
199. {
200. **if**(tree.elem[a].lchild>=0)
201. MidOrder(tree.elem[a].lchild,tree);
202. cout<<"\t"<<tree.elem[a].weight;
203. **if**(tree.elem[a].rchild>=0)
204. MidOrder(tree.elem[a].rchild,tree);
205. }

208. **void** EnCode(**const** HFTree &tree,**int** pos,**char** code[],**int** &len)
209. {
211. **int** ans=0,cnt=0;
212. **int** path[totalNumber];
213. **while**(pos!=tree.root)
214. {
215. //cout<<pos<<endl;
216. path[cnt++]=pos;
217. pos=tree.elem[pos].parent;
218. }
219. len=cnt;
220. **for**(**int** i=0;i<len;i++)
221. {
222. **if**(tree.elem[pos].lchild==path[cnt-1-i])
223. code[i]='0';
224. **else**
225. code[i]='1';
226. pos=path[cnt-1-i];
227. }
228. code[len]='\0';
229. **return**;
230. }
232. **void** DeCode(**char** value[],**char** tar[],**const** HFTree &tree,**char** result[])
233. {
234. **int** cnt=0, tmp=tree.root;
235. **for**(**int** i=0;tar[i];i++)
236. {
237. **if**(tar[i]=='0' && tree.elem[tmp].lchild>=0)
238. {
239. tmp=tree.elem[tmp].lchild;
240. }
241. **else** **if**(tar[i]=='1' && tree.elem[tmp].rchild>=0)
242. {
243. tmp=tree.elem[tmp].rchild;
244. }
245. **else**
246. {
247. i--;
248. result[cnt++]=tree.elem[tmp].data;
249. tmp=tree.root;
250. }
251. }
252. result[cnt++]=tree.elem[tmp].data;
253. result[cnt]='\0';
254. }
256. **int** main ()
257. {
258. HFTree tree;
259. **int** fr[]={4,7,5,2,9};
260. **char** value[]={'a','b','c','d','e'};
261. **char** code[totalNumber][500];
262. **int** codelen[totalNumber];
263. createHFTree(tree,value,fr,5);
265. ShowPreOrder(tree);
266. ShowMidOrder(tree);

269. cout<<endl;
270. **for**(**int** i=0;i<5;i++)
271. EnCode(tree,i,code[i],codelen[i]);
272. cout<<"明文\t:\t密文"<<endl;
273. **for**(**int** i=0;i<5;i++)
274. {
275. cout<<value[i]<<"\t:\t"<<code[i]<<endl;
276. }
278. cout<<endl;
279. **char** tar[]="11000111000101011";
280. **char** result[500];
281. DeCode(value,tar,tree,result);
282. cout<<tar<<"的明文是"<<result<<endl;// ecabceeb
283. }
284. 运行结果（程序运行时的结果说明或运行截图等）



六、总结（实验中遇到的问题、取得的经验、感想等）

1、使用最小堆优化了算法，在老师的ppt中，哈夫曼树的构造里寻找最小结点是用的历遍的算法，是O(n)的，而最小堆寻找最小结点是O(1)的，算上维护最小堆的时间，不超过堆的高度，是O(logn)的。最小堆在寻找最小结点上功能节约了很多时间。

2、结合第一点，现在的哈夫曼树构造的总时间复杂度是O(n\*logn)的，这是一个非常快的速度，而ppt上的复杂度时O(n^2)的，可以说最小堆太有用了。

3、密文的加密和解密都是依赖在树上的基础上的，其复杂度是O(len\*logn)和O（len）的，两个len分别是明文和密文的长度，n是明文所用的不同字符的个数。

4、最小堆可以很方便得改成最大堆。