**实验4-哈夫曼编码**

**问题分析**

**哈夫曼树构造**

* 哈夫曼编码的本实现用了贪心算法的理念：先对给定的若干点的权进行分割，让他们变为单个的森林（单个的树），随后在其中选择两个权值最小的节点生成新的节点，再将新生成的结点参与森林中进行筛选，直至无结点进行筛选，在此过程中均需要更新原节点的父节点索引值以及新节点的左子树索引和右子树索引。
* 针对上述选择两个权值最小的两个节点，此次实验先后试验了两种方法：
  + 自定义选择函数。具体功能为遍历哈夫曼树表并找到最小和次小节点的索引值。时间复杂度。
  + STL库中multiset容器。将哈夫曼树表中的节点放入multiset容器中。利用其自带的类似优先队列的功能找到索引值。时间复杂度。

综合比较时间复杂度、实现难度以及维护等因素采用方法二。

1. multiset<node> s;
2. **for** (**int** i = 1; i <= n; i++)
3. {
4. cin >> huffmantree[i].weight;
5. huffmantree[i].idx = i;
6. huffmantree[i].parent = 0;
7. s.insert(huffmantree[i]);
8. }

* 为达到题目中要求三：左儿子的权值>=右儿子的权值和要求四：相同权值w的两个字母x、y，先输入权值的字母x的Huffman编码长度不超过后输入权值的字母y的Huffman编码长度。，需对结构体采用双关键字排序。即先比较权值，若权值相等比较二者在哈夫曼树表中的索引值大小。

结构体设计如下：

1. **struct** node {
2. **public**:
3. **int** parent, lc, rc, weight, idx;
4. string code;
6. **bool** operator<(**const** node& p) **const** {
7. **if** (**this**->weight < p.weight)
8. **return** **true**;
9. **else** **if** (**this**->weight == p.weight)
10. {
11. **if** (**this**->idx > p.idx)
12. **return** **true**;
13. **else** **return** **false**;
14. }
15. **else** **return** **false**;
16. }
17. }huffmantree[100];

**哈夫曼编码构造**

* 对哈夫曼树采取自底向上搜索，即从叶节点向上寻找它的父亲节点，直至找到根节点

。每次寻找判断叶节点是父节点的左子树还是右子树。

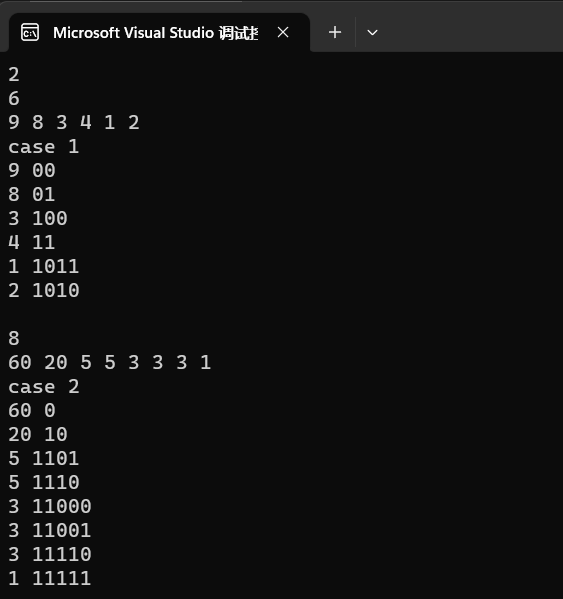
* + 对于右子树将其答案字符串尾部+1。
  + 对于左子树将其答案字符串尾部+0。

最后将答案字符串进行逆置。并将该字符串保存在每个节点的code变量中。

具体代码如下：

1. **void** createhuffmancode(node\* a, **int** num)
2. {
3. **for** (**int** i = 1; i <= num; i++)
4. {
5. **int** temp = i;//孩子的位置也要更新
6. string code;
7. code.clear();
8. **int** f = a[i].parent;
9. **while** (f != 0)
10. {
11. **if** (a[f].lc == temp)
12. {
13. code += '0';
14. }
15. **else** **if** (a[f].rc == temp)
16. {
17. code += '1';
18. }
19. temp = f;
20. f = a[f].parent;
21. }
22. reverse(code.begin(), code.end());
23. a[i].code = code;
24. }
25. }

**运行结果截图**



**设计调试中的问题**

* 在使用STL容器的auto迭代器时，遇到不可修改的左值问题，即无法通过(\*it).parent进行更新每一个节点的父节点索引值。

解决方法：先找到该迭代器当下节点的索引值，再根据该索引值在哈夫曼树表中查找。

1. huffmantree[(\*it).idx].parent = temp;

* 在调式时出现死循环，查询哈夫曼表，发现每次合并新节点时忘记将原节点父节点更新。

1. **while** (s.size() != 1)
2. {
3. auto it = s.begin();
4. huffmantree[temp].weight = (\*it).weight;
5. huffmantree[temp].rc = (\*it).idx;
6. huffmantree[(\*it).idx].parent = temp;
7. it++;
8. huffmantree[temp].weight += (\*it).weight;
9. huffmantree[temp].lc = (\*it).idx;
10. huffmantree[(\*it).idx].parent = temp;
11. ... ...
12. }

* 在测试时发现和案例输出不一样，不满足要求三。在检查结构体重载函数无误后，发现在加入新节点时应先记录右子树后记录左子树，因为需要左子树值大于右子树，而自定义的multiset是从小到大排序。

1. huffmantree[temp].rc = (\*it).idx;
2. huffmantree[temp].lc = (\*it).idx;

**实验体会**

此次实验相对前三个实验比较简单。但在实验过程中也曾遇到一些问题。从一开始自主编写哈夫曼树中取两个最小元素位置的函数到后来使用stl库中multiset容器。我不仅了解到了这些功能函数最底层的实现，也意识到应熟练stl库中的各种容器，遇到问题时才能更高效快速地解决。

在数据结构课程中曾学习过有关哈夫曼树的知识，但当时仅仅是对原理的了解，并没用进行实际操作。而本次实验，不仅让我复习了树的相关知识，也让我真正地实现了哈夫曼树及其编码的应用，在实现过程中的一些细节处理也让我对哈夫曼树有了更深地认识，

**程序源码**

1. #include<iostream>
2. #include<bits/stdc++.h>
3. **using** **namespace** std;
5. **struct** node {
6. **public**:
7. **int** parent, lc, rc, weight, idx;
8. string code;
10. **bool** operator<(**const** node& p) **const** {
11. **if** (**this**->weight < p.weight)
12. **return** **true**;
13. **else** **if** (**this**->weight == p.weight)
14. {
15. **if** (**this**->idx > p.idx)
16. **return** **true**;
17. **else** **return** **false**;
18. }
19. **else** **return** **false**;
20. }
21. }huffmantree[100];
23. **void** createhuffmancode(node\* a, **int** num);
24. **void** test(node\* a, **int** num);
25. multiset<node> s;
27. **void** creathuffmantree(**int** n)
28. {
29. **int** temp = 1;
30. **for** (**int** i = 1; i <= n; i++)
31. {
32. cin >> huffmantree[i].weight;
33. huffmantree[i].idx = i;
34. huffmantree[i].parent = 0;
35. s.insert(huffmantree[i]);
36. temp++;
37. }
38. **while** (s.size() != 1)
39. {
40. auto it = s.begin();
41. huffmantree[temp].weight = (\*it).weight;
42. huffmantree[temp].rc = (\*it).idx;
43. huffmantree[(\*it).idx].parent = temp;
44. it++;
45. huffmantree[temp].weight += (\*it).weight;
46. huffmantree[temp].lc = (\*it).idx;
47. huffmantree[(\*it).idx].parent = temp;
48. huffmantree[temp].idx = temp;
49. huffmantree[temp].parent = 0;
50. s.erase(s.begin());
51. s.erase(s.begin());
52. s.insert(huffmantree[temp]);
53. temp++;
54. }
55. createhuffmancode(huffmantree, n);
56. }
57. **void** test(node\* a, **int** num)
58. {
59. **for** (**int** i = 1; i <= num \* 2 - 1; i++)
60. {
61. cout<< a[i].idx <<' ' << a[i].weight << ' ' << a[i].lc << ' ' << a[i].rc << ' ' << a[i].parent << endl;
62. }
63. }
65. **void** createhuffmancode(node\* a, **int** num)
66. {
67. **for** (**int** i = 1; i <= num; i++)
68. {
69. **int** temp = i;//孩子的位置也要更新
70. string code;
71. code.clear();
72. **int** f = a[i].parent;
73. **while** (f != 0)
74. {
75. **if** (a[f].lc == temp)
76. {
77. code += '0';
78. }
79. **else** **if** (a[f].rc == temp)
80. {
81. code += '1';
82. }
83. temp = f;
84. f = a[f].parent;
85. }
86. reverse(code.begin(), code.end());
87. a[i].code = code;
88. }
89. }
90. **int** main()
91. {
92. **int** t;
93. cin >> t;
94. **for**(**int** i = 1;i <=t;i++)
95. {
96. s.clear();
97. **int** n;
98. cin >> n;
99. creathuffmantree( n);
100. cout << "case " << i << endl;
101. **for** (**int** i = 1; i <= n; i++)
102. {
103. cout << huffmantree[i].weight << ' ' << huffmantree[i].code << endl;
104. }
105. cout << endl;
106. }
107. }