****

**实 验 报 告**

**实验课程： 数据结构与算法实验**

**实验题目： 哈夫曼树的建立与编码**

**任课老师： 吴之旭老师**

**学生姓名： 殷骢睿**

**学 号： 5509121041 \_**

**专业班级： 2021级人工智能实验班 \_**

**2022 年 12 月 9日**

** 南昌大学实验报告**

学生姓名： 殷骢睿 学号： 5509121041 专业班级： 人工智能211班

实验类型：□ 验证 □ 综合 ■ 设计 □ 创新 实验日期： 2022.12.9 实验成绩：

1. **实验目的**

1、掌握哈夫曼树的建立

2、掌握哈夫曼编码方式

1. **实验内容**

**具体请完成如下功能：**

(1) 先定义单个结点的信息，包括父节点，左右权值和左右孩子节点；

(2) 设置select函数，在下标为1到i-1的范围找到权值最小的两个值的下标，其中s1的权值小于s2的权值；

(3) **构建函数CreateHuff(HuffmanTree& HT, DataType\* w, int n)，void HuffCoding(HuffmanTree& HT, HuffmanCode& HC, int n)，函数功能为构建哈夫曼树和哈夫曼编码。**

**三、实验要求**

(1) 程序要添加适当的注释，程序的书写要采用**缩进格式**。

(2) 程序要具在一定的**健壮性**，即当输入数据非法时，程序也能适当地做出反应，如查找的内容不存在时，要给予提示等等。

(3) 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。

(4) 根据实验报告模板详细书写实验报告,在实验报告中给出**流程图**。

(5) **源程序和实验报告打包。**

**四、设计实现哈夫曼树的建立与编码主要实验步骤及程序分析**

**4.1思路设计**

实现霍夫曼编码的方式主要是创建一个二元树和其节点。这些树的节点可以储存在阵列里，阵列的大小为符号数的大小n，而节点分别是终端节点（叶节点）与非终端节点（内部节点）。

一开始，所有的节点都是终端节点，节点内有三个栏位：

1.符号（Symbol）

2.权重（Weight、Probabilities、Frequency）

3.指向父节点的链接（Link to its parent node）

而非终端节点内有四个栏位：

1.权重（Weight、Probabilities、Frequency）

2.指向两个子节点的链接（Links to two child node）

3.指向父节点的链接（Link to its parent node）

基本上，我们用'0'与'1'分别代表指向左子节点与右子节点，最后为完成的二元树共有n个终端节点与n-1个非终端节点，去除了不必要的符号并产生最佳的编码长度。

过程中，每个终端节点都包含著一个权重，两两终端节点结合会产生一个新节点，\*\*新节点的权重是由两个权重最小的终端节点权重之总和，并持续进行此过程直到只剩下一个节点为止。

实现霍夫曼树的方式有很多种，可以使用优先伫列简单达成这个过程，给与权重较低的符号较高的优先顺序，演算法如下：

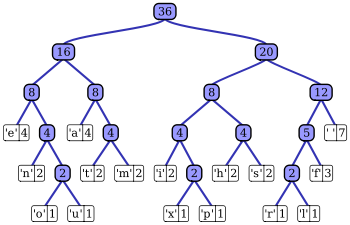
1.把n个终端节点加入优先伫列，则n个节点都有一个优先权Pi，1 ≤ i ≤ n

2.如果伫列内的节点数>1，则：

1. ​从伫列前端移除两个最低权重的节点
2. ​将（1）中移除的两个节点权重相加合成一个新节点
3. 加入第二个伫列

3.最后在第一个伫列的节点为根节点

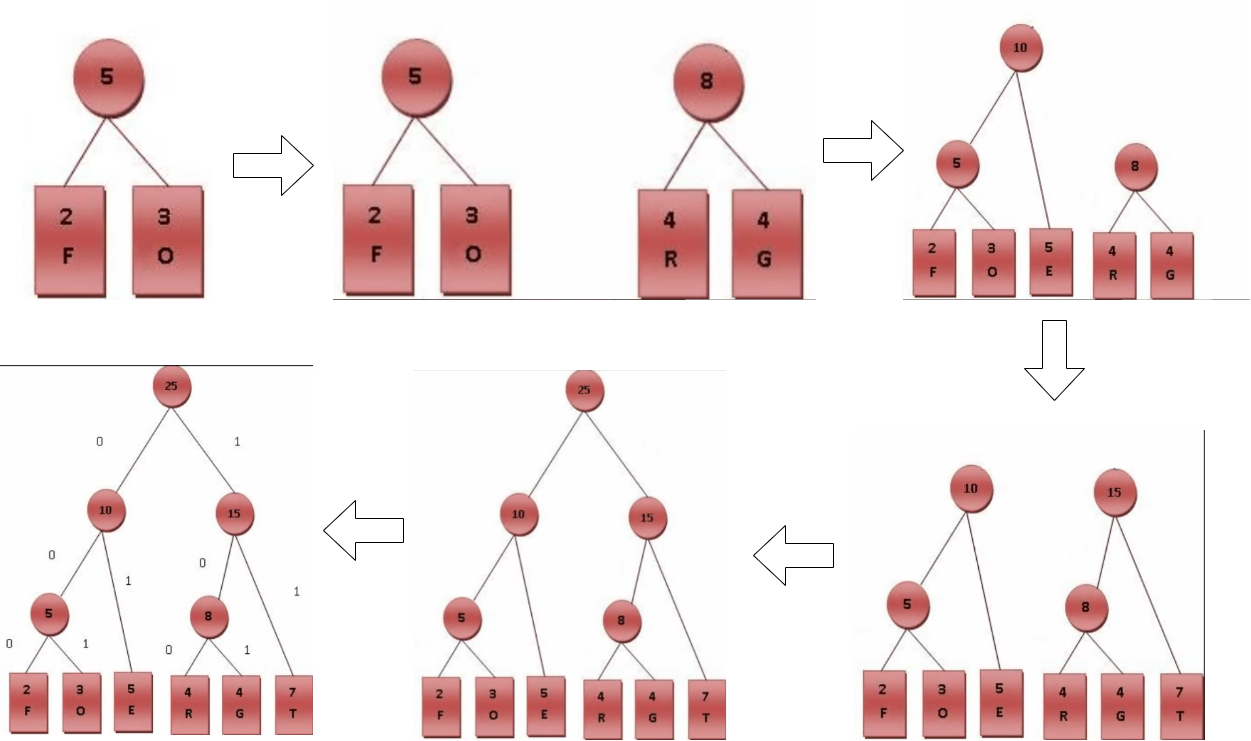
虽然使用此方法比使用优先伫列的时间复杂度还低，但是节点必须依照权重大小加入伫列中，如果节点加入顺序不按大小，则需要经过排序，则至少花了O（nlogn）的时间复杂度计算。



**图1 霍夫曼树图**

| **字母** | **频率** | **编码** |
| --- | --- | --- |
| space | 7 | 111 |
| a | 4 | 010 |
| e | 4 | 000 |
| f | 3 | 1101 |
| h | 2 | 1010 |
| i | 2 | 1000 |
| m | 2 | 0111 |
| n | 2 | 0010 |
| s | 2 | 1011 |
| t | 2 | 0110 |
| l | 1 | 11001 |
| o | 1 | 00110 |
| p | 1 | 10011 |
| r | 1 | 11000 |
| u | 1 | 00111 |
| x | 1 | 10010 |

**表1 霍夫曼编码表**

****

**图2 霍夫曼树的六个步骤（以编码FORGET为例）**

**4.2主要具体步骤（完整代码见附录）**

**4.2.1 定义单个节点的信息，包括父节点，左右权值和左右子节点**

1. **struct** HuffNode {
2. **int** weight;       // 权值
3. **int** parent;       // 父节点下标
4. **int** lchild;       // 左孩子节点下标
5. **int** rchild;       // 右孩子节点下标
6. };

**4.2.2 设置select函数，在下标为1到i-1的范围找到权值最小的两个值的下标，其中s1的权值小于s2的权值**

详细相关步骤如源码注释所示：

1. // 找到下标为1到i-1的权值最小的两个节点的下标
2. **void** select(HuffNode huff[], **int** i, **int**& s1, **int**& s2) {
3. // 初始化最小值和次小值
4. s1 = s2 = 0;
5. **for** (**int** j = 1; j < i; j++) {
6. // 如果当前节点未被选中并且权值比最小值小
7. **if** (huff[j].parent == 0 && huff[j].weight < huff[s1].weight) {
8. s2 = s1;
9. s1 = j;
10. }
11. // 如果当前节点未被选中，权值比最小值大，比次小值小
12. **else** **if** (huff[j].parent == 0 && huff[j].weight < huff[s2].weight) {
13. s2 = j;
14. }
15. }
16. }

**4.2.3 构建函数CreateHuff(HuffmanTree& HT, DataType\* w, int n)**

详细相关步骤如源码注释所示：

1. // 构建哈夫曼树
2. **void** CreateHuff(HuffmanTree& HT, DataType\* w, **int** n) {
3. // 初始化每个节点
4. **for** (**int** i = 1; i <= 2 \* n - 1; i++) {
5. **if** (i <= n) {
6. HT[i].weight = w[i];
7. }
8. **else** {
9. HT[i].weight = 0;
10. }
11. HT[i].parent = 0;
12. HT[i].lchild = 0;
13. HT[i].rchild = 0;
14. }
16. // 循环构建哈夫曼树
17. **for** (**int** i = n + 1; i <= 2 \* n - 1; i++) {
18. // 选择下标为1到i-1的权值最小的两个节点的下标
19. **int** s1, s2;
20. select(HT, i, s1, s2);
22. // 构建新结点
23. HT[i].weight = HT[s1].weight + HT[s2].weight;
24. HT[i].lchild = s1;
25. HT[i].rchild = s2;
26. HT[s1].parent = i;
27. HT[s2].parent = i;
28. }
29. }

**4.2.4 构建函数HuffCoding(HuffmanTree& HT, HuffmanCode& HC, int n)**

先临时存储编码，再循环构建哈夫曼编码，主要思想是从叶子节点往上寻找根节点，然后根据左右子节点的判断来构建编码。

详细相关步骤如源码注释所示：

1. **void** HuffCoding(HuffmanTree& HT, HuffmanCode& HC, **int** n) {
2. // 临时存储编码
3. **char**\* cd = **new** **char**[n];
4. cd[n - 1] = '\0';
6. // 循环构建哈夫曼编码
7. **for** (**int** i = 1; i <= n; i++) {
8. **int** start = n - 1;
9. // 从叶子节点开始往上找到根结点
10. **for** (**int** c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) {
11. // 如果当前结点是右孩子，编码为1，否则为0
12. **if** (HT[f].lchild == c) {
13. cd[--start] = '0';
14. }
15. **else** {
16. cd[--start] = '1';
17. }
18. }
20. // 将构建好的编码存储在数组中
21. HC[i] = **new** **char**[n - start];
22. strcpy(HC[i], &cd[start]);
23. }
24. // 释放临时存储编码的内存
25. **delete**[] cd;
26. }

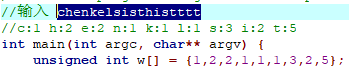
**4.2.5 主函数int main(int argc, char\*\* argv)**

调用上述头文件，实现调用接口的运行和打印对应的数据功能：

1. #include <iostream>
2. #include "huffmantree.h"
4. **using** **namespace** std;
5. /\* run this program using the console pauser or add your own getch, system("pause") or input loop \*/
6. //输入 chenkelsisthistttt
7. //c:1 h:2 e:2 n:1 k:1 l:1 s:3 i:2 t:5
8. **int** main(**int** argc, **char**\*\* argv) {
9. unsigned **int** w[] = {1,2,2,1,1,1,3,2,5};
10. HuffmanTree HT;
11. HuffmanCode HC;
12. HuffmanCoding(HT,HC,w,9);
13. **for**(**int** i=1;i<=9;i++){
14. cout<<"HC"<<i<<"的赫夫曼编码"<<HC[i]<<endl;
15. }
16. **return** 0;
17. }

**4.3程序结果**

**输入：字符串 char chenkelsisthistttt，手动统计相关数据**



**图3 输入样例示意图**

**输出：生成字符所对应的Huffman编码**



**图4 程序结果图**

**五、编写代码时遇到的问题及实验心得体会**

**遇到的问题：**

1. 写霍夫曼树的代码时，开始不知道如何找到每个数据项在哪个节点。后面为了解决这个问题，我发现可以给每个节点赋予一个唯一的编号，然后通过查询这个编号来找到对应的节点。
2. 另一个问题是实现霍夫曼树的解码。这个问题可以通过建立一个哈夫曼编码表来解决，该表中包含每个数据项对应的哈夫曼编码。然后，当解码时，可以通过查询哈夫曼编码表来找到对应的数据项。
3. 总的来说，编写霍夫曼树代码会遇到的问题主要是如何实现哈夫曼树的建立和解码。通过给每个节点赋予唯一的编号，建立哈夫曼编码表等技巧，可以有效解决这些问题。

**实验心得体会：**

(1) 在编写霍夫曼树代码的实验过程中，我体会到了霍夫曼树的优越性能。通过使用霍夫曼树，我们可以有效地将数据进行编码，使得编码后的数据长度更短，从而节省存储空间和传输带宽。

(2) 此外，通过实验，我还发现霍夫曼树的构建过程相对比较复杂，需要较多的算法技巧才能实现。但是，一旦构建完成，就可以方便地进行数据编码和解码，从而大大提高数据处理的效率。

(3) 编写霍夫曼树代码的实验给我留下了深刻的印象，让我更加理解和欣赏霍夫曼树的优越性能。这次编程坚决没有上网搜索现成的方案，锻炼了自己的意志力和克服困难的能力，很感谢老师的这次考验，这可能是最后一次做数据结构后的作业了，感觉在此期间代码能力提升非常明显，精通数据结构不是一朝一夕所能完成的事情，需要锲而不舍的努力。我相信我有信心与毅力将这些知识理解透彻，提高自己的综合能力，并且将数据结构知识熟练运用于未来的科研和互联网工程当中。

