# 四川师范大学 实验报告

学期： 2024 至 2025 第一学期 实验成绩：

课程名字：《程序设计基础——数据结构(C语言版)》 专业： 信息与计算科学

班级： 2023 级 9 班 实验编号： 07

实验项目： 实验七 指导老师： 冯山

姓名： 刘智恒 学号： 2023060522

**一、实验题目**

Huffman树和Huffman编码算法

**二、实验目的及要求**

掌握Huffman树的编码方法和算法实现

**三、实验内容：(类C算法的程序实现，任选其二)**

1.建立Huffman树编码算法的表示结构。设计并实现以下算法：对给定的输入字符串进行Huffman编码，并输出字符串的编码方案。

2.设计并实现以下算法：对实现给定字符串，利用Huffman编码算法对其进行编码，并输出编码结果。

3.设计并实现以下算法：对给定的Huffman编码串，利用已有的Huffman编码方案对其进行解码，并输出解码结果字符串。

**四、实验准备**

1.计算机设备;

2.程序调试环境的准备，本实验采用**Microsoft Visual Studio**环境;

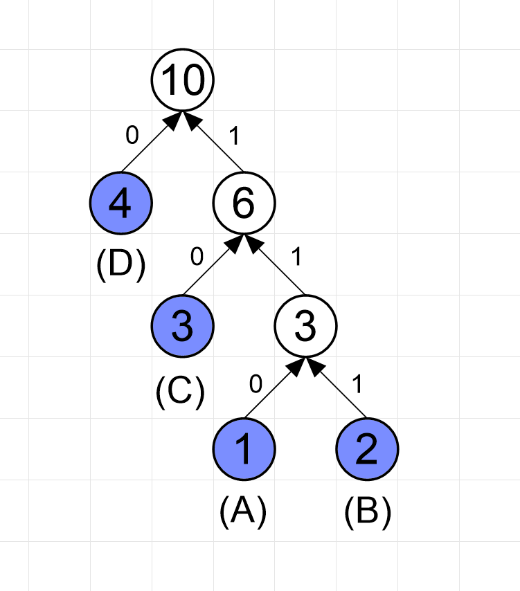
3.实验内容的算法分析与代码设计与分析准备；

4.实验源程序**Exp\_7(1)**准备。

5.实验测试用例：

**（1）给定字符串：ABCBCDCDDD**

**构建的Huffman树如下图所示（圈内表示权值）：**

****

**图4.1**

**由此可得各个字符的赫夫曼编码：**

**A：110**

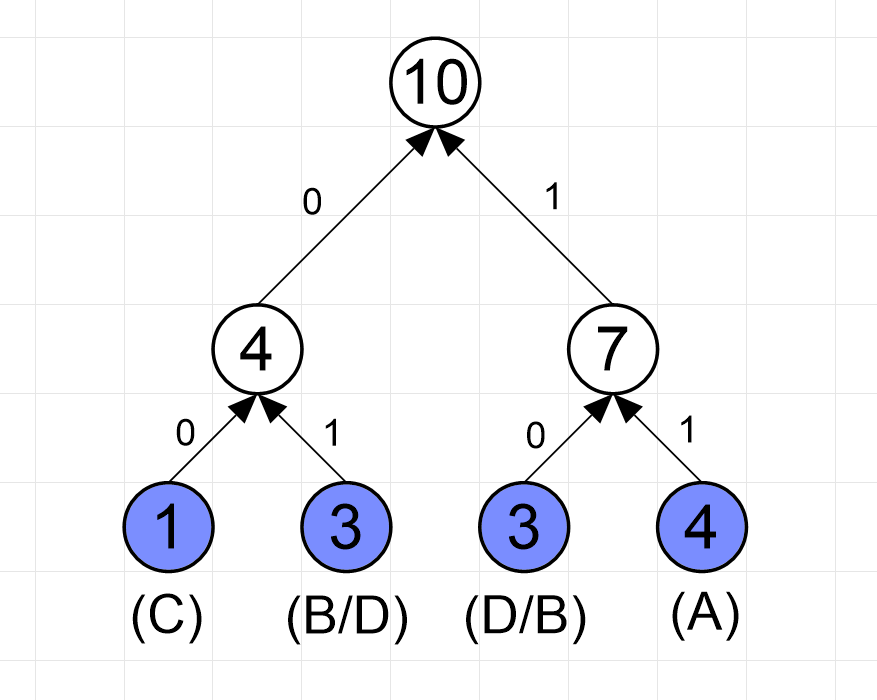
**B：111**

**C：10**

**D：0**

**（2）给定字符串：CADABDBABAD**

**构建的Huffman树如下图所示（圈内表示权值）：**



**图4.2**

**由此可得各个字符的赫夫曼编码：**

**A：11**

**B：01/10**

**C：00**

**D：10/01**

**五、实验内容**

**（一）问题分析**

（1）首先给出**Huffman树**的定义：Huffman树（**赫夫曼**）又称最优树（通常讨论**最优二叉树**），是一类**带权路径长度最短**的树。结点的带权路径长度为**从该结点到树根之间的路径长度与结点上权的乘积**；树的带权路径长度为**树中所有叶子结点的带权路径长度之和**，通常记作：

由此带权路径长度**WPL**最小的二叉树称作**最优二叉树**或**Huffman树**。

（2）**赫夫曼编码**则是以设计**总长最短**的电文为背景的实际问题。设计电文总长最短的编码即为**二进制前缀编码**，所谓**前缀编码**即：**任一个字符的编码都不是另一个字符的编码的前缀**。故以n种字符**出现的频率作为权值**，设计一棵**Huffman树**，从而得到的二进制前缀编码便称为**赫夫曼编码**。

（3）由于Huffman树中**没有度为1的结点**，则一棵有**n个叶子结点**的Huffman树共有**2n-1个结点**，故可以存储在一个大小为**2n-1**的**一维数组**中。在构成Huffman树之后，为了求出**编码**则需要从叶子结点出发**走一条从叶子到根的路径**；而为了**译码**则需要从根出发走一条**从根到叶子的路径**。故对每个结点而言，**既需要知道双亲的信息，又需要知道孩子结点的信息**。由此，Huffman树和赫夫曼编码的存储结构表示如下：

**typedef struct{**

**unsigned int weight;** //结点权值

**unsigned int parent, lchild, rchild;** //双亲，左孩子，右孩子

**}HTNode, \*HuffmanTree;** //动态分配数组存储Huffman树

**typedef char\*\* HuffmanCode;**  //动态分配数组存储赫夫曼编码表

（4）各项操作基本分析：

**[1]统计字符的频率**：对于输入的一个字符串，其中各个字符的频率以其**出现的次数**为准，可以用一整型数组**frequency**来记录频率。对于每个输入的字符**input[i]**，则对应其频率：**frequency[input[i]]++**。最终以每个字符的**频率作为权值**来构建Huffman树。

**[2]构建Huffman树**：在此之前需将每个结点的双亲域以及左右孩子域的值置为0，然后选取所有权值（结点）中**最小的两个权值作为左右（孩子）节点**，其双亲结点的权值则为该**两个最小权值的和**；对于已选取的权值，其双亲结点权值**必不为0**，故在剩下的**双亲域的值为0的权值**（结点）中继续选取最小的两个权值（结点），**逐步向上**构建Huffman树。直到Huffman树所有的**2\*n-1个节点均有权值**。

**（注：选取的两个最小权值节点作为子树的左右孩子时，若顺序不同，则构建的Huffman树不同，最终的编码结果也不同）**

**[3]求赫夫曼编码**：在构建完Huffman树后，约定一棵子树上的**左分支表示字符‘0’，右分支表示字符‘1’**；**从根节点到各个叶子（字符）节点**的路径求出编码结果，以字符串形式输出。

**（二）算法描述**

（1）**Status InitHuffmanTree(HuffmanTree \*HT, int n)**;

//本算法用于初始化Huffman树：动态分配内存空间以存储2\*n-1个Huffman树节点；将所有节点的双亲节点、左孩子和右孩子索引初始化为0。

（2）**void SelectTwo(HuffmanTree HT, int n, int \*s1, int \*s2)**;

//本算法用于选择两个权值最小的节点：遍历树中所有权重未被使用的节点（parent == 0），找到两个权重最小的节点并记录它们的索引。

（3）**Status CreateHuffmanTree(HuffmanTree HT, int n)**;

//本算法用于创建Huffman树：从当前节点数n开始，逐步构建Huffman树直到达到2\*n-1，在每一步中选择两个权重最小的节点，并将它们作为新节点的左右孩子，新的节点的权重为这两个最小节点权重之和。

（4）**void CreateHTCode(HuffmanTree HT, HuffmanCode \*HC, int n)**;

//本算法用于生成Huffman编码：遍历每个叶子节点（即实际字符），从该叶子节点向上追溯到根节点。在追溯过程中，如果当前节点是其双亲节点的左孩子，则在编码中添加'0'；如果是右孩子，则添加'1'。最后将构建的编码存储在Huffman编码数组中。

（5）**void PrintHuffmanCode(HuffmanCode HC, int n)**;

//本算法用于输出Huffman编码：遍历Huffman编码数组，打印每个字符对应的Huffman编码。

（6）**Status FreeSpace(HuffmanTree \*HT, HuffmanCode \*HC, int n)**;

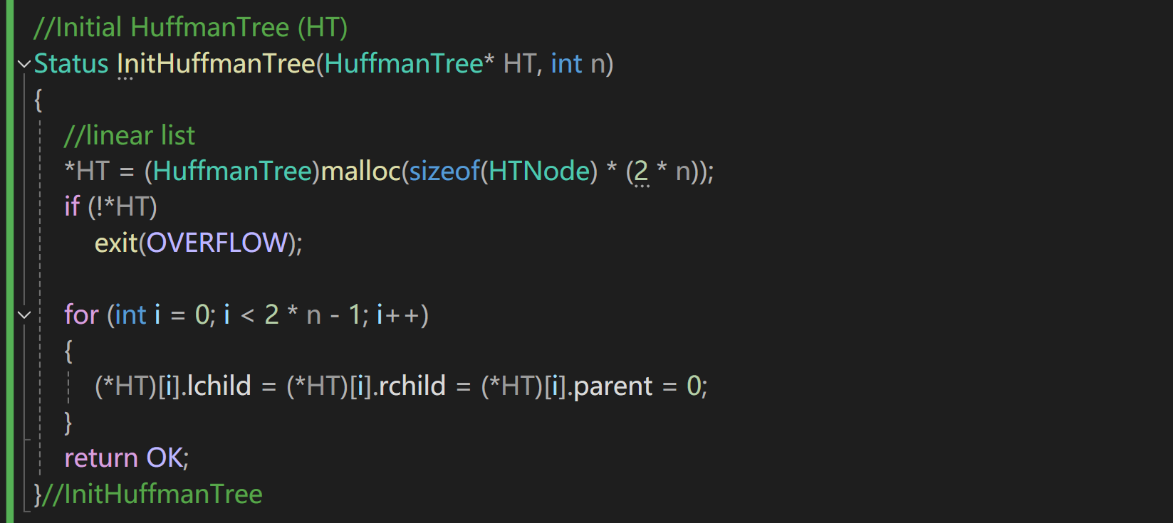
//本算法用于释放已分配的内存：释放分配的Huffman树和Huffman编码数组的内存，防止内存泄漏。

**（三）程序代码**

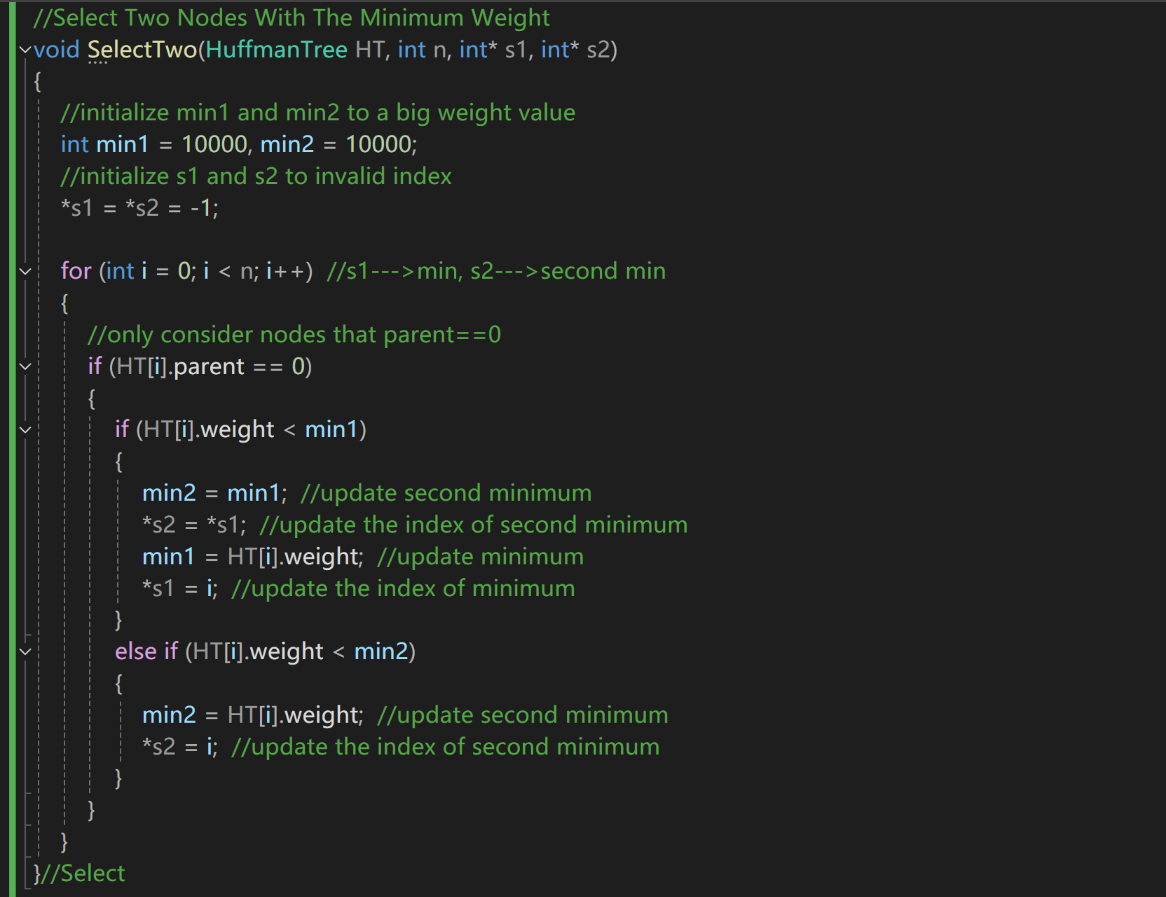
根据以上对**实验内容**的算法描述，求解的程序代码如下：



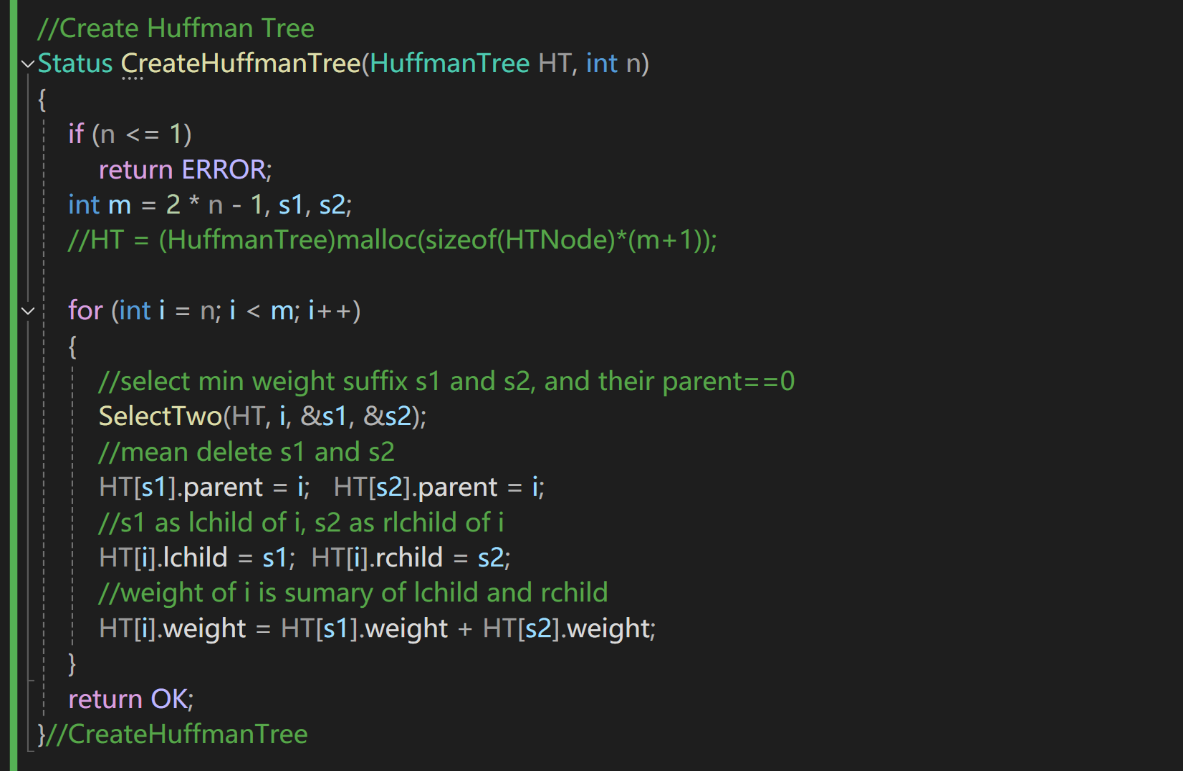
**注：此处只展示主要算法的程序代码图。**



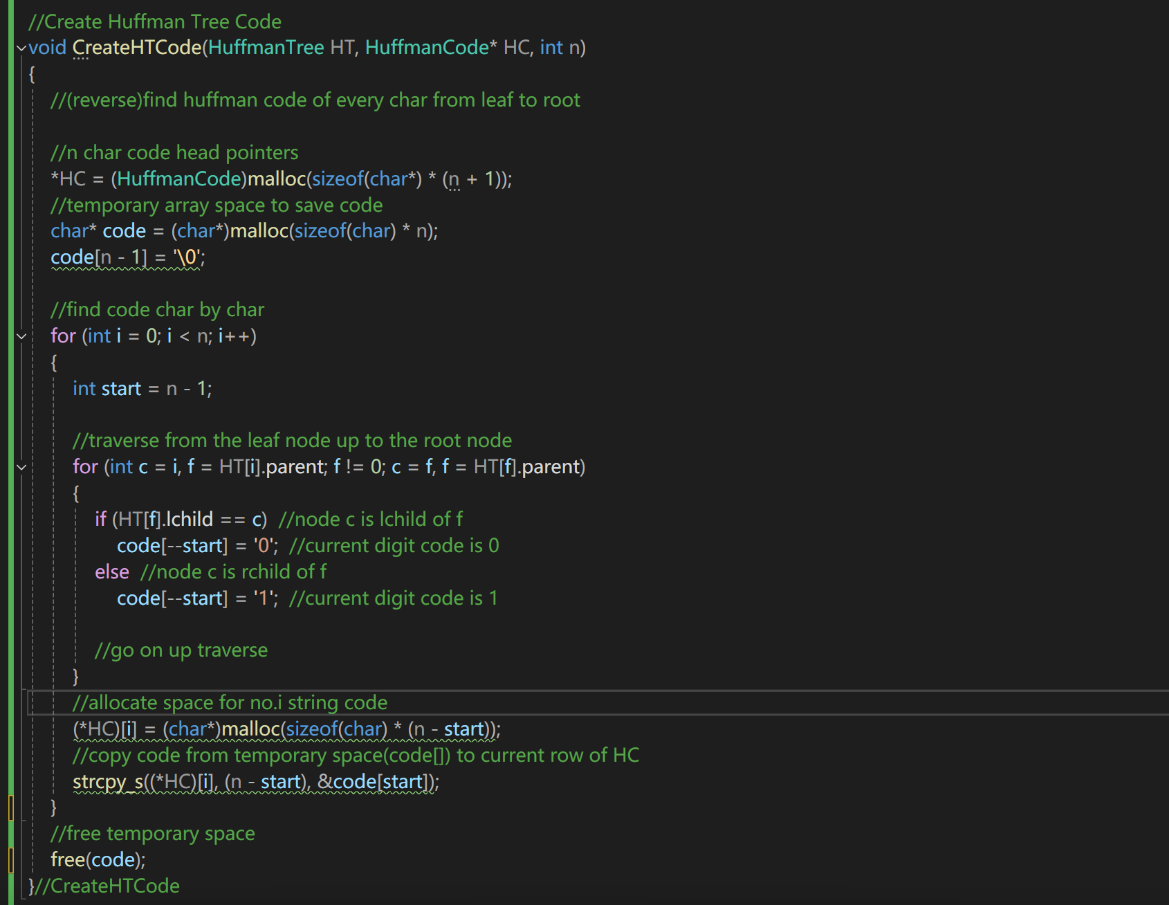
**图5.1初始化Huffman树**



**图5.2选择两个最小权值节点**



**图5.3创建Huffman树**

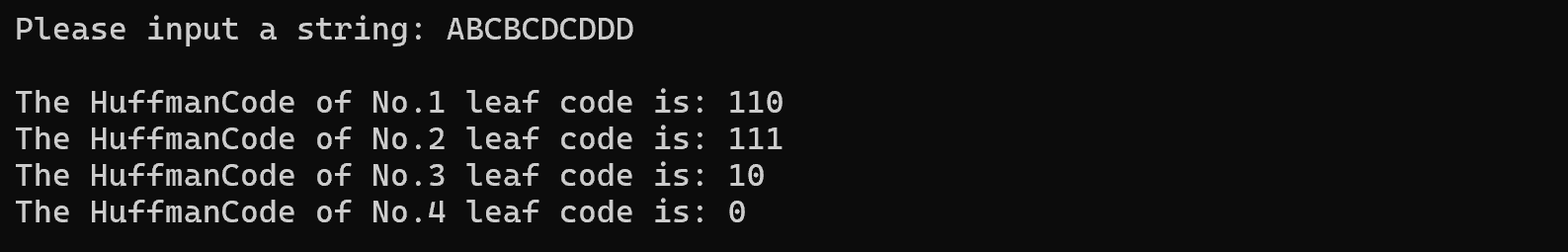


**图5.4生成Huffman编码**

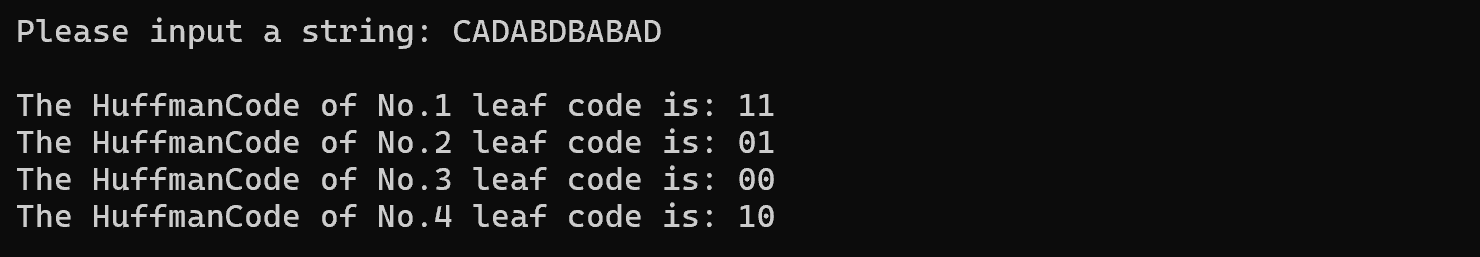
**六、实验结果**

**（一）结果呈现：**

1.根据以上**实验内容**的程序代码运行后进行多次测试，以下是根据**实验内容测试用例**进行测试后的结果示意图：



**图6.1实验测试用例（1）结果**



**图6.2实验测试用例（2）结果**

**（二）结果分析：**

**1.基本分析**

（1）将结果与测试用例中的Huffman树示意图对比后，结果无误。需要注意的是，此时是以选取的两个最小权值节点中：**以最小的权值作为子树左孩子，次小的权值作为子树右孩子**从而得到的Huffman编码结果。根据示意图不难发现：构造的顺序是**由右下到左上**。

（2）若**调换顺序**，即：**以次小的权值作为子树左孩子，最小的权值作为子树右孩子**，则此时得到的Huffman编码结果必然有所不同。易知：此时的构造顺序是**由左下到右上**。**以测试用例1作测试**，不难得到各个字符的Huffman编码结果如下：

**A：001**

**B：000**

**C：01**

**D：1**

**2.时间复杂度分析**

**【InitHuffmanTree】—— 初始化Huffman树**

使用一个循环初始化了**2\*n-1**个节点的双亲、左子树和右子树指针，故该函数的时间复杂度是**O(n)**。

**【SelectTwo】—— 选择两个最小权值节点**

在每次选择两个权重最小的节点时，该函数会遍历当前的节点数**n**。在最坏情况下，函数的时间复杂度为**O(n)**。

**【CreateHuffmanTree】—— 创建Huffman树**

第一个循环从**n**到**2\*n-1**（共**n-1**次），在每次迭代中调用**SelectTwo**函数；由于每次选择需要**O(n)**的时间，故整个函数的时间复杂度是**O(n^2)**。

**【CreateHTCode】—— 生成Huffman编码**

对于每个字符（一共**n**个），都会遍历从叶节点到根节点的路径，最坏情况下路径长度为**O(n)**，故该函数的时间复杂度是**O(n^2)**。

**【PrintHuffmanCode】—— 输出Huffman编码**

遍历每个字符并输出对应的Huffman编码，故该函数的时间复杂度为**O(n)**。

**综上所述，时间复杂度主要是由CreateHuffmanTree和CreateHTCode造成的，故最终的时间复杂度为：O(n^2)**。

**3.空间复杂度分析**

**【Huffman树**】

**HuffmanTree HT**需要存储**2\*n-1**个节点，每个节点占用常数空间，因此哈夫曼树的空间复杂度是**O(n)**。

**【Huffman编码】**

**HuffmanCode HC**需要指向**n**个字符串，每个字符串在最坏情况下也包含长度为**n**的哈夫曼编码。因此，Huffman编码的空间复杂度是**O(n^2)**（**O(n)**的指针加上**O(n^2)**的字符数组）。

**【输入字符串和频率数组**】

输入字符**input[1000]**的空间复杂度是**O(1)**，因为大小是固定的。频率数组**frequency[MAX\_CHAR]**的空间复杂度是**O(1)**，同样也是由于大小固定。

**综上所述，总空间复杂度主要由Huffman编码提供，因此最终的空间复杂度为**：**O(n^2)。**

**七、实验总结**

1.对于用户输入的字符串，应该更对输入进行**有效性检查和处理**，比如过滤特殊字符。

2.当前采用的选取两个最小节点的方法复杂度较高（**O(n^2)**），可以考虑使用**优先队列**来优化选取过程，降低时间复杂度至**O(n\*logn)**。

3.在**CreateHTCode**中，可以直接通过**数组**方式存储哈夫曼码，而不必在构造时先存储在临时的字符数组中再复制。