**Instruction and Explaination**

**EN**

**I, Explaination**

***1, Main idea explaination***

“CanonicalHuffman” is the same as Normal Huffman coding.

1, It builds a Huffman tree based on Frequency with the same algorithm with normal Huffman.

2, Then the program will sort the characters based on their code length(how deep they are in the Huffman Tree) from small to big. If the length are same, they will be sorted based on their ascii code(from small to Big). At the end, we will receive and array Header, which contain character and the length of code, sorted

3, The code will be generated with Canonical Huffman algorithm:

- Header[0].char ’s code = Header[0].length ‘0’  
example: if the first char have code length is 3, then their code will be “000”

-int mask=0

- loop until reach the last element of head

if i>0: mask++, then left shift mask with the amount equal to

(Header[i].length-Header[i-1].length)  
Then the code will be equal to first “Header[i].length” bits of mask.

Example: Header[1].length=4, then mask=1<<(4-3)=……0102, then the code will be: “0010”

4, About the Header:  
4.1, The number of characters is small then 30 – mode 1:

The header wil contain: (mode)(max\_len)(usedchar)(number\_char\_of\_len\_1,....,number\_char\_of\_max\_len)(char1)(char2)...(char[usedchar])

-mode: 1  
-max\_len: the maximum len of code  
-usedchar: the number of char which is in the input file  
-number\_char\_of\_len\_i: amount of character which have the same code length i

-(char1)(char2)...(char[usedchar]): Header[i].char

4.2: Used char >= 30 – mode 2

The Header will contain:  
(mode)(max\_len)(len\_of\_char[0],len\_of\_char[1],...len\_of\_char[255])

Mode: 2

Max\_len: max\_len: the maximum len of code

(len\_of\_char[0],len\_of\_char[1],...len\_of\_char[255]): the code length of char[0] to char[255], this will be compressed.

5, maincode:

Convert the main text to the string of 1 and 0, then convert that string into string of char.

***2, Code Explaination***

We will have the class Huffman, which contain Encode and Decode functions

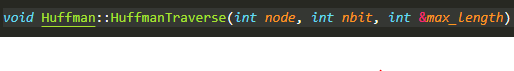


*Into the Encode Phase:*

 Print frequency Table  
Read the file, then save the data of file and the Frequency of character

Find 2 elements with smallest frequency. i,j are min elements, nodeN is the max index of node.

Create huffman Tree with normal Huffman algorithm, the return is root node.

Go through the tree and save the code length(how deep) of each char into BitCode Bit[256]

 Call the HuffmanTraverse and get the max\_length(max\_len)

 Create Codebook, buiding Header Head[]

 Convert Code to string of 1,0

Convert string which contain 1,0 into string compress, 8 character in source = 1 character in destination

Reverse of the ConvertStringToBit function

: Encode to the “filename” file

It will call the huffman tree building, the Codebook building, then buid a Header, and the last is main code. It’s will work like in the Main idea explaination

*Into the Decode Phase:*  
It will read data from “filename”, first it will read the Header to define which mode was used:  
 -If in the mode 1, it will read max\_len, UsedChar, then the other of Header, then rebuild Code book.

Example: maxlen=4, usedchar=8, and array of len is 0143, and the array of char is (abcdefgh)=> length of a=2, length of bcde=3, length of fgh=4.

Then based on length, it will rebuild codebook with CanonicalHuffman algorithm.

-If mode is 2, it’s will read the max\_len, and the remain of Header, then we will have the exact length of each character from char=0 to char=255 save in

Bitcode Bit[256], then just call CreateBitcodeTable to rebuild Codebook.

Then it will read the maincode, put the max\_len in the end of the main\_code, then return that string.

 Reverse of FromCharToCode.

It will first call the DecodeInput function, then extract the max\_len and “remain” in the end of maincode, then call BitToString to receive then string of 1 and 0, then call FromCodeToChar to get the plain text.

**II, Instruction**

In the main fuction, you will be asked that you want to Encode or not. Enter Y or y to Encode. Then it’s will asked you to enter the plaintext filename. And then the output filename. It will automatically add .huff extension to output filename.

In decode phase, enter Y or y to decode, you have to enter the inputfile (full name, include .huff)  
Then enter the file you want to save the result.

**РУ**

**I, Объяснение**

***1, Объяснение Основной Идеи***

«Каноническое кодирование Хаффмана» аналогично нормальному кодированию Хаффмана.

1. Он строит дерево Хаффмана на основе частоты по тому же алгоритму, что и обычный Хаффман.

2. Затем программа отсортирует символы по длине кода (насколько глубоко они находятся в дереве Хаффмана) от маленького до большого. Если длина одинакова, они будут отсортированы на основе их кода ascii (от маленького до большого). В конце мы получим и массив Header, который содержит символ и длину кода, отсортированный

3, Kод будет сгенерирован с помощью канонического алгоритма Хаффмана:

- Код Header [0] .char = Header[0] .length ‘0’

Пример: если длина кода первого символа равна 3, то его код будет «000».

-int mask = 0

- петля до последнего элемента головы

если i> 0: mask ++, то mask сдвига влево с величиной, равной

(Header[i] .length-Header[i-1] .length)

Тогда код будет равен первым «Header [i] .length». битам маски

Пример: Header[1] ​​.length = 4, затем mask = 1 << (4-3) = …… 0102, тогда код будет: «0010»

4, О заголовке:

4.1, Количество символов меньше 30 - режим 1:

Header будет содержать: (режим) (max\_len) (usedchar) (number\_char\_of\_len\_1, ...., number\_char\_of\_max\_len) (char1) (char2) ... (char [usedchar])

-режим: 1

-max\_len: максимальная длина кода

-usedchar: количество символов во входном файле

-number\_char\_of\_len\_i: количество символов с одинаковой длиной кода i

- (char1) (char2) ... (char [usedchar]): Header[i] .char

4.2: Используемый символ >=30 - режим 2

Header будет содержать:

(режим) (max\_len) (len\_of\_char [0], len\_of\_char [1], ... len\_of\_char [255])

Режим: 2

Max\_len: max\_len: максимальная длина кода

(len\_of\_char [0], len\_of\_char [1], ... len\_of\_char [255]): длина кода от char [0] до char [255], это будет сжато.

5, Основной код:

Преобразуйте основной текст в строку из 1 и 0, затем преобразуйте эту строку в строку символов.

***2, Объяснение кода***

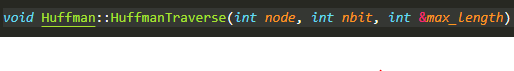
У нас будет класс Huffman, содержащий функции кодирования и декодирования.

*В фазу кодирования:*

 Печать Таблица частоты   
 Прочтите файл, затем сохраните данные файла и частоту символов

Найдите 2 элемента с наименьшей частотой. i, j - минимальные элементы, nodeN - максимальный индекс node.

 Создайте дерево Хаффмана с помощью обычного алгоритма Хаффмана, возвращение - корневой узел.

 Просмотрите дерево и сохраните длину кода (глубину) каждого символа в BitCode Bit [256]

 Вызовите HuffmanTraverse и получите max\_length (max\_len) Создайте Кодовую книгу, создав Header Head[]

 Преобразовать код в строку 1,0

 Преобразовать строку, которая содержит 1,0 в сжатую строку, 8 символов в источнике = 1 символ в месте назначения

 Реверс функции ConvertStringToBit

: : Закодировать в файл «filename». Он вызовет построение дерева Хаффмана, здание кодовой книги, затем построит заголовок, и последний будет основным кодом. Это будет работать как в объяснении основной идеи

*В фазу декодирования:*  
 Он будет читать данные из «filename», сначала он будет читать заголовок, чтобы определить, какой режим был использован:

-Если в режиме 1, он будет читать max\_len, UsedChar, затем остальные в заголовке, а затем перестраивать книгу кодов.

Пример: maxlen = 4, usedchar = 8, а массив len равен 0,1,4,3, а массив char равен (a, b, c, d, e, f, g, h) => длина a = 2, длина b, c, d, e = 3, длина f, g, h = 4.

Затем, исходя из длины, он перестроит кодовую книгу с помощью алгоритма CanonicalHuffman.

-Если режим равен 2, он будет читать max\_len и оставшуюся часть заголовка, тогда у нас будет точная длина каждого символа от char = 0 до char = 255, сохраненная в

Bitcode Bit [256], затем просто вызовите CreateBitcodeTable, чтобы восстановить кодовую книгу.

Затем он прочитает основной код, поместит max\_len в конец main\_code, а затем вернет эту строку. Реверс FromCharToCode.

 Сначала он вызовет функцию DecodeInput, затем извлечет max\_len и «remain» в конце основного кода, затем вызовет BitToString для получения затем строки из 1 и 0, затем вызовет FromCodeToChar для получения простого текста.

**II, Инструкция**

В основной функции вас спросят, хотите вы кодировать или нет. Введите Y или Y для кодирования. Затем вам будет предложено ввести имя файла в виде открытого текста. И затем имя выходного файла. Он автоматически добавит расширение .huff к выходному имени файла.

На этапе декодирования введите Y или y для декодирования, вы должны ввести входной файл (полное имя, включая .huff)

Затем введите файл, в который хотите сохранить результат.