
CS29003 ALGORITHMS LABORATORY

ASSIGNMENT 5

Date: 22th Aug, 2019

Important Instructions

1. **List of Input Files:** log.txt, encode.txt, decode.txt
 2. **Output Files to be submitted:** ROLLNO_A5_output.txt, ROLLNO_A5_P1.c/.cpp, ROLLNO_A5_P2.c/.cpp, ROLLNO_A5_P3.c/.cpp, ROLLNO_A5_P4.c/.cpp
 3. **THERE WILL BE PART MARKING.**
 4. Files must be read using file handling APIs of C/C++. Reading through input redirection isn't allowed.
 5. You are to stick to the file input output formats strictly as per the instructions above. Failing to do so might cause you to lose marks.
 6. Submission through **.zip files aren't allowed.**
-

Huffman Coding

Intro

For communicating through digital mediums, we must encode messages into bit streams first. In such a system, the sender needs an encoder for encoding text messages into bit streams, and the recipient, on receiving this bit stream, uses a decoder for reconstructing the original message. Today we will see such an algorithm called Huffman Coding, which can encode text messages into bit streams of minimal length. The algorithm is based on a binary-tree frequency-sorting method that allows to encode any message sequence into a bit stream and reassemble any encoded message into its original version without losing any data. When you compare between fixed-length binary encoding and Huffman coding, the later has the least possible expected message length (under certain constraints).

Objectives of this assignment

This assignment has four parts. In part I, you will have to estimate frequencies of every character from the log file. In part II you will have to compute the Huffman codes for each character. In part III you will have to build the encoder based on Huffman codes. In the final part you will have to build a decoder to retrieve the data back.

Part I - Observing Frequency Distribution

As a first step towards building an efficient prefix code for communication, you would need to gather symbol probabilities first. Thankfully, you have access to a log of previously communicated messages as a text file. The format of the file is as follows,

FILE: *log.txt*

15
the
quick
brown
fox
jumps
over
a
lazy
dog
the
five
boxing
wizards
jump
quickly

The first line contains the number of words (messages) recorded in *log.txt*. From this file you need to read in all the words and build a symbol (character) frequency list. **As output of this part you should print the frequency list as the very first line of your output.txt.** The set of symbols is limited to lowercase alphanumeric characters. And you must print them in proper order, a to z and then 0 to 9.

Note: For any symbol not occurring in the log, set their frequency to 0 and include them in all your computations ahead.

Part II - Finding the Huffman Coding

Step 1

Now that we have our frequency list ready, we will implement the Huffman Coding Algorithm. We will start with a sorted linked list of symbol frequencies. See Figure 1 for an example. When facing equal count, for **conflict resolution** use the priority order $\{a, \dots, z\} < \{0, \dots, 9\}$.

Data structure

The basic data structure should follow the template below. We will explain the use of the pointers 'left' and 'right' soon.

```
class Node{
    char *symbol;
    int frequency;
    Node *next;
    Node *left;
    Node *right;
}
```

Initially, all left and right pointers set to null.

Step 2

Remove the two lowest frequency nodes from the linked list and merge them together to create a SuperNode N_i (i: an incremental id starting from 1). This supernode N_i should have the two merged nodes as children, as shown in Figure 2, with the one with lower frequency as left child. (Any conflict, if they appear, should be resolved using the priority order $N_i < \{a, \dots, z\} < \{0, \dots, 9\}$.)

- ▷ $N_i.\text{left}$ = lowest frequency node
- ▷ $N_i.\text{right}$ = the other node
- ▷ $N_i.\text{frequency}$ = sum of the frequencies of its children
in this case, $11 + 23 = 34$

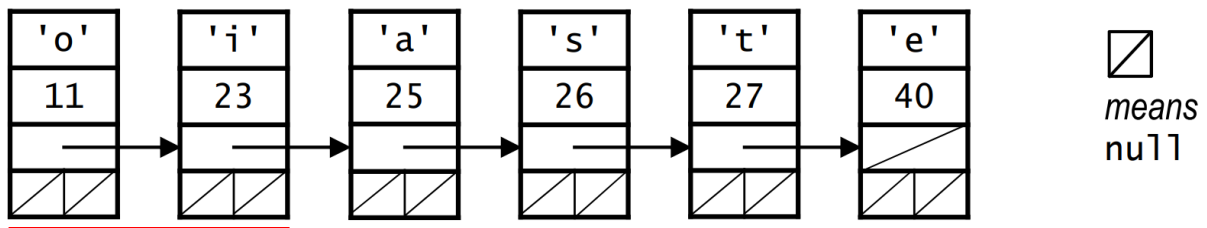


Figure 1: Linked List of symbols

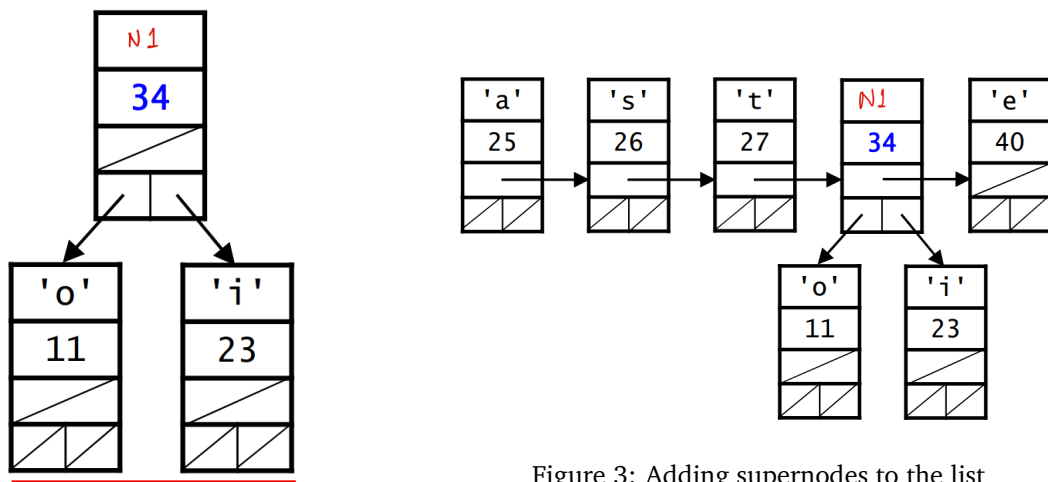


Figure 3: Adding supernodes to the list

Figure 2: Merging nodes to create supernodes

Step 3

Add the supernode to the list of nodes (maintaining sorted order). Check Figure 3

Step 4

Repeat steps 2 and 3 until there is only a single node in the list and the final structure thus obtained is a tree (Figure 4). We will call this the **Huffman tree**. Note that the single node left in the linked list is the root of this tree.

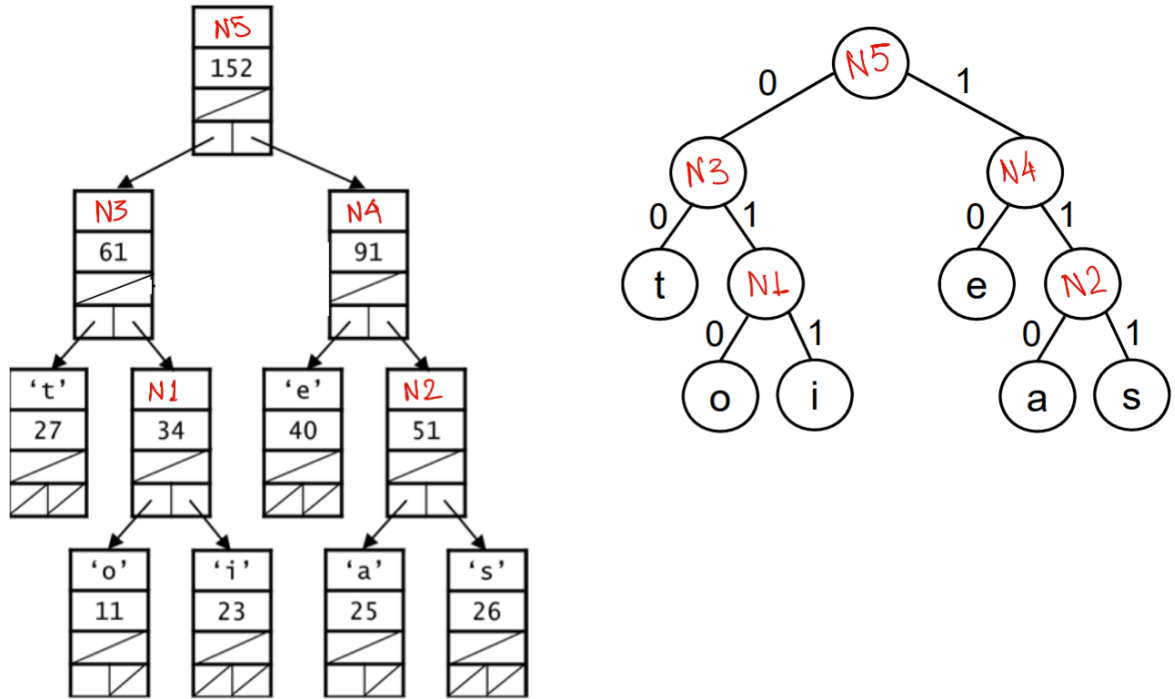


Figure 4: Huffman Tree

Step 5

Obtaining the Huffman Codes: Once the Huffman tree is built, you can obtain the Huffman code for each symbol based on the path from the root to the leaf node holding that symbol. If we denote a path (of length k) by $v_0v_1\dots v_k$ (root: v_0 , leaf: v_k), the corresponding Huffman code for v_k .symbol will be $1_{r(0,1)}, 1_{r(1,2)}, \dots, 1_{r(k-1,k)}$. $1_{r(i,i+1)}$ is the indicator¹ function for “whether v_{i+1} is a right child of v_i ”. For example, in Figure 4 Huffman code for ‘t’ will be ‘00’, but it will be ‘010’ for ‘o’. For finding all such paths use TreeTraverse on the Huffman tree and obtain the Huffman codes *on-the-fly*.

Algorithm 1: TreeTraverse

Input: T: Root Node of Tree, C: prefix code, H: Array for storing all Huffman Codes

if T is a leaf **then**

 H[T.symbol] = C;

else

 TreeTraverse(N.left, concat(C,0));

 TreeTraverse(N.right, concat(C,1));

end

Outputs: As output of Part II, you should append the following lines to your output.txt file (created in Part I).

- ▷ The preorder traversal of the Huffman Tree. e.g. N5, N3, t, N1, o, i, N4, e, N2, a, s for the example given in Fig 4.
- ▷ The next 36 lines should contain the Huffman code or prefix code for **all alphanumeric characters** in alphabetic order, in the following format. During initialization of the linked list use count of 0 for symbols unseen in log.txt.

a 0000

¹Indicator Function: $1_{r(0,1)} = 1$ if v_1 is right child of v_0 , else 0

```
b 01000
c 00100
d 10111
e 0110
...
...
y 11000
z 01010
0 110100100
1 110100111111
...
9 110100101
```

Part III - Encoding

Using the prefix codes found above for each alphanumeric character, encoding new messages is pretty easy. From step 5 in Part I, you should have an array *H* of strings containing the Huffman codes for each alphanumeric characters. For any given new word(message), you need to read it char-by-char (symbols) and encode each to them in left to right order. To test out your implementation, we will need you to encode the sample messages from the *encode.txt* file. **Output the encoded messages to *output.txt* as usual, with one encoded message per line.**

Notation: In Algorithm 2 and 3, *H*[*c*] denotes the Huffman code for character or symbol *c*.

Algorithm 2: Encode a message

Input: *H*: Huffman Codes, *encodedMessage*

Output: *R*: Encoded Message

R = "";

for *c* in *encodedMessage* **do**

 | *R* = concat(*R*, *H*[*c*])

end

```
// Decode function template
void encode(char* message, char* encodedMessage){
    // YOUR LOGIC HERE
    // Output is saved to encodedMessage
}
```

Part IV - Decoding

Although we have this great and efficient encoder built, we still need to be able to decode received messages also. That's where the decoder comes into place. Given an encoded message of 0's and 1's, the decoder will convert it back to the original message. Since the encoding scheme is a prefix code, we can start reading from left to right and apply a greedy matching scheme for decoding symbols. **Same as before, output the decoded messages in the *output.txt*.**

Algorithm 3: Decode an encoded message

Input: S: Symbol Alphabet, H: Huffman Codes, encodedMessage

Output: R: Decoded message

cache = "";

R = "";

for char *b* in encodedMessage **do**

 // b can be '0' or '1' only;

 cache = concat(cache, b);

for *s* ∈ S **do**

if cache == *H[s]* **then**

 R = concat(R,s);

 reset cache;

end

end

end

```
// Decoder function template
void decode(char* encodedMessage, char* decodedOutput){
    // YOUR LOGIC HERE
}
```

Example

A complete set of sample input and output files is made available through moodle, please check attachments.

FILE: *log.txt*

```
15
the
quick
brown
fox
jumps
over
a
lazy
dog
the
five
boxing
wizards
jump
quickly
```

FILE: *encode.txt*

3
zombie
without
legs

FILE: *decode.txt*

3
0011000000100100100110110
1111010011111100011011110
0101111000011011110100110011011110

FILE: *output.txt*

a=3,b=2,c=2,d=2,e=4,f=2,g=2,h=2,i=5,j=2,k=2,l=2,m=2,n=2,o=5,p=2,q=2,r=3,s=2,t=2,u=4,v=2,w=2,
x=2,y=2,z=2,0=0,1=0,2=0,3=0,4=0,5=0,6=0,7=0,8=0,9=0

N35 N33 N29 o N21 N11 N10 N9 N8 N7 N6 N5 N4 N3 N2 N1 0 1 2 3 4 5 6 7 8 9 b c N12 d f N30 N22

N13 g h N14 j k N23 N15 l m N16 n p N34 N31 N24 N17 q s N18 t v N25 N19 w x N20 y z N32 N27

e u N28 i N26 a r

a 11110

b 001001

c 00101

d 00110

e 1100

f 00111

g 01000

h 01001

i 1110

j 01010

k 01011

l 01100

m 01101

n 01110

o 000

p 01111

q 10000

r 11111

s 10001

t 10010

u 1101

v 10011

w 10100

x 10101

y 10110

z 10111

0 0010000000000000

1 0010000000000001

2 001000000000001

3 00100000000001

4 001000000001

5 00100000001

6 0010000001

7 001000001

8 00100001

9 0010001

101110000110100100111101100

1010011101001001001000110110010

0110011000100010001

dobby

avada

kedavda