CS29003 ALGORITHMS LABORATORY
ASSIGNMENT 4 (Binary Trees)
Date: 1st Oct, 2020

Important Instructions

1. **List of Input Files:** log.txt, encode.txt, decode.txt. Example files are provided in Moodle and at the end of this PDF.
2. **Files to be submitted:** ROLLNO_GroupNO_A4.c/.cpp
3. Your code should output a file 'output.txt' when run. Again, example files are provided in Moodle.
4. Files must be read using file handling APIs of C/C++. Reading through 'input redirection'/'scanf' isn't allowed.
5. You are to stick to the file input output formats strictly as per the sample files. Failing to do so might cause you to lose marks.
6. ONLY PART I AND PART II ARE COMPULSORY FOR TODAY'S ASSIGNMENT. PART III AND IV ARE BONUS.
7. IF YOU SUBMIT THE BONUS PARTS, THOSE WILL BE GRADED AND CONSIDERED ONLY IN THE CASE OF SOMEONE FALLING ON A CLOSE GRADE BOUNDARY IN THIS COURSE AT THE END OF THE SEMESTER.
8. Everything must be submitted within lab hours.

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*

```
16
the
quick
brown
fox
jumps
over
a
lazy
dog
the
five
boxing
wizards
jump
quickly
ten10
```

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.

Example output:

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

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,

$$a < b < ... < z < 0 < 1 < ... < 9 \tag{1}$$

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 (SuperNodes)

Remove the two lowest frequency nodes from the linked list and merge them together to create a SuperNode Ni (i: an incremental id starting from 1). This supernode Ni should have the two merged nodes as children, as shown in Figure 2, with the one with lower frequency as left child.

- ▷ Ni.left = lowest frequency node
- ▷ Ni.right = the other node
- ▷ Ni.frequency = sum of the frequencies of its children
  in this case, 11 + 23 = 34

Any conflict, if they appear, should be resolved using the priority order

$$N1 < N2 < ... < Ni < a < b < ... < z < 0 < 1 < ... < 9 \tag{2}$$

PRIORITY ORDERS MENTIONED IN EQUATION 1 AND EQUATION 2 ARE VERY IMPORTANT FOR THE OUTPUTS TO MATCH. IF THE SAME PRIORITY ORDERS ARE NOT USED, THE RESULTS WILL NOT MATCH WITH OURS AND MARKS WILL BE DEDUCTED.

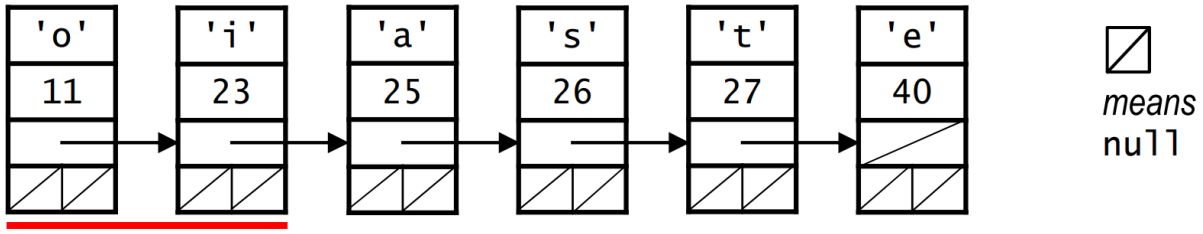


Figure 1: Linked List of symbols

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.

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 $v_0 v_1 ... v_k$ (root:v_0, leaf:v_k), the corresponding Huffman code for $v_k.\mathtt{symbol}$ will be $1_{r(0,1)}, 1_{r(1,2)}, ..., 1_{r(k-1,k)}$. $1_{r(i,i+1)}$ is the indicator[1] 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',

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

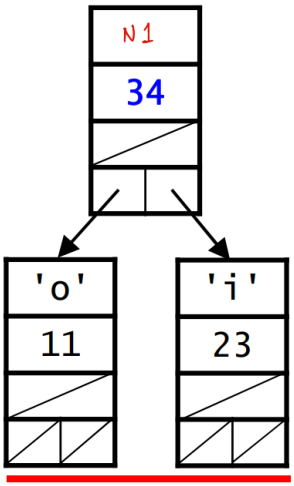


Figure 2: Merging nodes to create supernodes

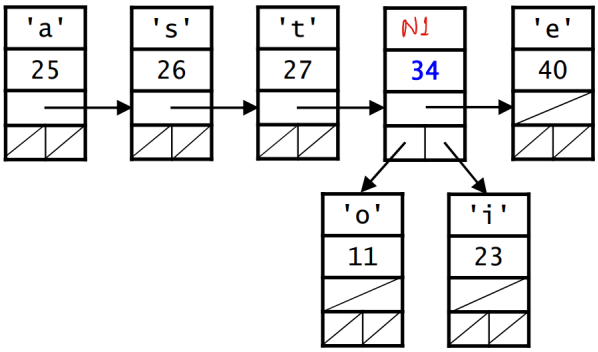
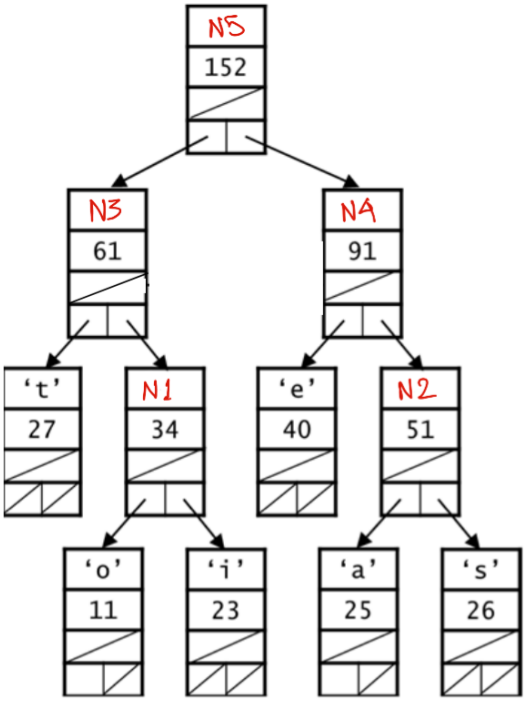


Figure 3: Adding supernodes to the list



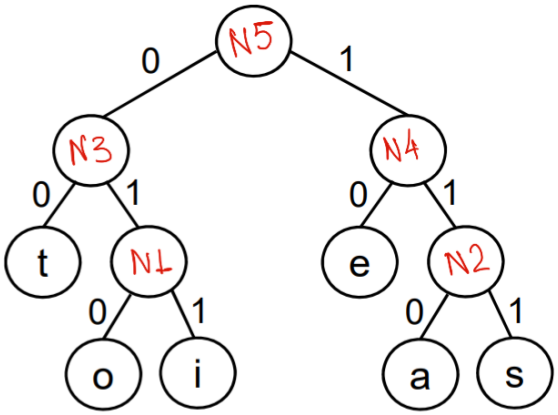


Figure 4: Huffman Tree

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).

▷ Print the **preorder traversal** of the Huffman Tree along with the frequency value for each node. e.g. for the Huffman Tree given in Fig 4, output should be,

```
N5(152),N3(61),t(27),N1(34),o(11),i(23),N4(91),e(40),N2(51),a(25),s(26)
```

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

Part III - Encoding

**** BONUS AND EVALUATED ONLY WHEN YOU ARE AT A GRADE BOUNDARY.**

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*

```
16
the
quick
brown
fox
jumps
over
a
lazy
dog
the
five
boxing
wizards
jump
quickly
ten10
```

FILE: *encode.txt*

```
3
zombie
without
legs
```

FILE: *decode.txt*

```
3
001011110000110001110011
101111000010111001010111
0101011000010110111110000001010111
```

```
a=3,b=2,c=2,d=2,e=5,f=2,g=2,h=2,i=5,j=2,k=2,l=2,m=2,n=3,o=5,p=2,q=2,r=3,s=2,t=3,u=4,v=2,w=2,
x=2,y=2,z=2,0=1,1=1,2=0,3=0,4=0,5=0,6=0,7=0,8=0,9=0
N35(69),N33(31),N29(15),N21(7),t(3),N10(4),N9(2),N8(1),N7(0),N6(0),N5(0),N4(0),N3(0),N2(0),
N1(0),2(0),3(0),4(0),5(0),6(0),7(0),8(0),9(0),0(1),1(1),b(2),N22(8),N11(4),c(2),d(2),N12(4),
f(2),g(2),N30(16),N23(8),N13(4),h(2),j(2),N14(4),k(2),l(2),N24(8),N15(4),m(2),p(2),N16(4),
q(2),s(2),N34(38),N31(17),N25(8),N17(4),v(2),w(2),N18(4),x(2),y(2),N26(9),u(4),N19(5),z(2),
a(3),N32(21),N27(10),e(5),i(5),N28(11),o(5),N20(6),n(3),r(3)
a 10111
b 00011
c 00100
d 00101
e 1100
f 00110
g 00111
h 01000
i 1101
j 01001
k 01010
l 01011
m 01100
n 11110
o 1110
p 01101
q 01110
r 11111
s 01111
t 0000
u 1010
v 10000
w 10001
x 10010
y 10011
z 10110
0 0001001
1 000101
2 00010000000000
3 00010000000001
4 0001000000001
5 000100000001
6 00010000001
7 0001000001
8 000100001
9 00010001
10110111001100000111101110 0
10001110100000100011101010000 0
0101111000011101111
dobby
avada
kedavda
```