CSCI 6620 / 2226 Data Structures Spring 2016

P6: From a HuffTree to a Huffman Code to a Coded File

The Huffman Tree This tree is the result of the heap operations in program 5. In this assignment, you will use the tree to build a compression code for the original file.

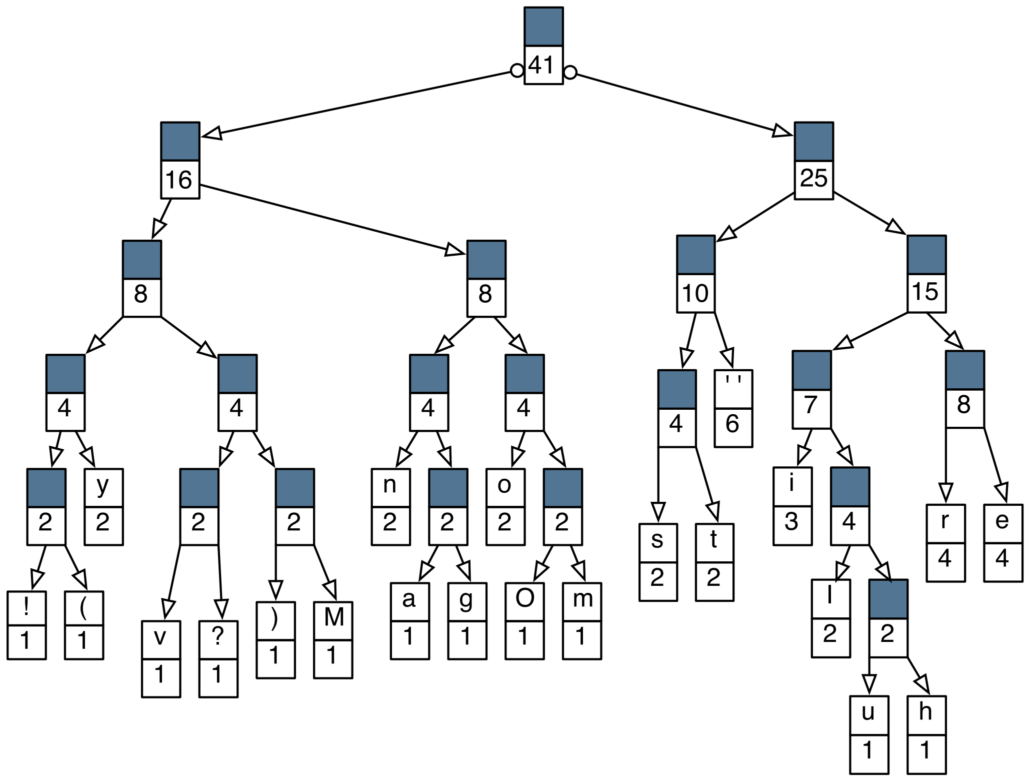

Figure 1: The Huffman tree, spread out neatly.

Step 4: Creating the Huffman Code

We can create a code by traversing the tree:

- Represent the code table as an STL `map<char, string>`.
- Use a string variable, *code*, to hold the digits of the code, as they develop. Initialize *code* to the empty string.
- Every left-branch in the tree corresponds to a 0 in the code, and every right-branch corresponds to a 1.
- Traverse the tree recursively, in depth-first order. Before a recursion on the left son, push a '0' onto the end of *code*; when the call returns, remove that '0'. Before a recursion on the right son, push a '1' onto the end of *code*; when the call returns, remove that '1'.
- When you reach a leaf node, a series of 0's and 1's is stored in *code*, which becomes the code for the letter stored in the leaf node. Add a new pair of < letter, code > to the map.

The table below shows the codes that would be generated from the tree above, in order of generation.

!	0	0	0	0	0	O	0	1	1	1	0	
(	0	0	0	0	1	m	0	1	1	1	1	
y	0	0	0	1		s	1	0	0	0		
v	0	0	1	0	0	t	1	0	0	1		
?	0	0	1	0	1	space	1	0	1			
)	0	0	1	1	0	i	1	1	0	0		
M	0	0	1	1	1	l	1	1	0	1	0	
n	0	1	0	0		u	1	1	0	1	1	0
a	0	1	0	1	0	h	1	1	0	1	1	1
g	0	1	0	1	1	r	1	1	1	0		
o	0	1	1	0		e	1	1	1	1		

This kind of code has some very special properties:

- It is a variable length code – common letters have shorter representations than uncommon ones.
- No code is a prefix of the code for any other letter. This enables us to decode messages.
- When a message is encoded, the bits are packed into bytes. Every bit counts. The last byte is padded with 0 bits, if necessary.

Step 5: Using the Huffman Code to Compress the Text

Let us now express the first part of our original message in this custom-made code:

Morals rule everything! (Or is it money?)

To do this easily, you will need to define a class called BitOutStream. This class will implement a buffering scheme for you so that you can give it a string, and it will take the bits corresponding to 1's and 0's and pack them into bytes. Completed bytes will be written to a binary output stream. There will be a separate writeup on this concept.

Begin the actual compression process by reopening the input file or seeking to its beginning. Then read the characters from the file one at a time, making sure NOT to skip whitespace. All characters that are in the file must be encoded. Execute this loop until end of file:

- Read a character.
- Use it to look up the corresponding code in the map.
- The code is a series of '0' and '1' chars. We need to convert each '0' to a 0 bit and each '1' to a 1 bit.
- Send the code (a string) to your BitOutStream to do the conversion from characters to bits and pack the code-bits into bytes that can be written to a normal output stream.
- At end of file, call BitOutStream::flush() to pad the last partial byte with 0 bits and write it to the output file. Then write out a byte containing the number of padding bits that were used.

Here is the encoding of the first 12 bytes of the message. The letters are in the left column, their codes in the middle column, and the packed 8-bit sections of the message are shown in the right column. Four 0 bits had to be added to the end to fill up the last byte.

M	00111	0011 1011
o	0110	0111 0010
r	1110	
a	01010	1011 0101
l	11010	
s	1000	0001 0111
' '	101	
r	1110	1011 0110
u	110110	
l	11010	1101 0111
e	1111	1101 0000
' '	101	

Here is the first part of the message written on one line. The second line shows how the bits correspond to the original letters The original 12 bytes have been reduced to 7 bytes plus a padding count. Thus, the compressed size is only 75% of the original size. Not bad.

```
00111011 01110010 10110101 00010111 10110110 11010111 11010000 00000100
M    o    r    a    l    s    ''r  u    l    e    ''        4
```

Step 6: Decompressing the File

This is not part of your assignment. It is provided for your information only.

To decompress a message, we need the Huffman tree and a BitInputStream that will read bytes from the compressed file and feed them to our program one bit at a time. We start decoding at the beginning of the message and at the root of the code tree:

- Read a bit and and go one level down the tree, left or right.
- When the program reaches a leaf node, output the letter at that node.
- Start again at the root of the code tree. Read another bit.

For example, in this instance, we read 00111, so we go left-left-right-right-right and hit an M. We output the M, and start back at the root. Next, 0110 (left-right-right-left) takes us to a little-o Then 1110 (right-right-right-left) gives us an 'r', and so on. After finding the second space, only 0000 and the padding bit-count (4) is left in the file. So the zeros are padding bits and are not part of the message. The BitInputStream will detect this condition and report that the decompression is finished successfully.