CSCI 6620 Fall 2018

Program 6: The Huffman Code

1 The Goals and the Purpose.

1. To use a heap to build a Huffman tree.
2. To implement Phase 3 of a Huffman encoding application.

Program 5, this assignment, and Program 7 form one large project to generate and apply a Huffman code. It is essential to debug this part before going on to the rest of the project. You will reuse all of the modules you created for Program 5. Be sure they are debugged!

2 From Heap to Huffman Code: Due November 1, 2018

The HuffTree class starts with the output of P5, a heap of Node pointers. It processes that heap until the Nodes have been incorporated into a Huffman code tree. Then it converts that tree to a code stored in a map, which is an efficient representation of the code to use for encoding a file.

Additions to the Node class.

- Write a second Node constructor that takes two Node*s as parameters (the left and right sons of the new node). Use the Node* parameters to initialize the pointer fields of the Node object. Set the char field to any distinctive and highly visible char value: you will never use it but you need to be able to see it on a printout. Set the frequency field to the sum of the frequencies of the left and right sons.
- You may need a default constructor for Node. If so, use `=default`.
- Define a proper destructor that deletes the left and right sons.
- You will not need getter functions. *Please tell me if you think you do.*
- Write a `debug()` function that displays all four fields of a Node with no newlines. The pointers will be printed in hex if you use <<.

Additions to the HuffHeap class. In P5, three of the five essential heap functions were implemented: `upHeap(), downHeap()`, and `heapify()`. In this project, we implement the remaining two functions: `remove()`, and `add()`, both illustrated in the Program 6 Supplement.

- The `add()` It places a new Node in the first unused position of the heap, and increments the Heap size. Then it calls `upHeap()` to push the new Node into heap order.
- The `remove()` function copies slot 1 of the heap into a temporary so it can later be returned. Then the last Node in the heap is moved to slot 0, and the heap size is decremented. Finally, `downHeap()` to push the new slot-1 Node into heap order.

The HuffTree class.

- Write a `heapToTree()` function with one parameter, a pointer to a HuffHeap. On each iteration, remove two Nodes from the heap, combine the data from the two nodes into a new node, and add it back into the heap.

  You start with N Nodes in the heap. On each iteration, the heap is shortened twice, then lengthened once. So reducing it to a tree will take N iterations. When the last iteration is completed, return the pointer to the Node in slot 0.

- Write a `print()` function with two parameters, a Node* and string. The first time you call this function, the parameters will be the root of the Huffman tree and the empty string.

  The function will do a recursive INFIX treewalk: recursively print the left side, print the node, then recursively print the right side. Each time you make a recursive call, append four spaces to the string parameter. This will print the tree in an indented format with virtually no effort.

- Write a `mapCode()` function with three parameters, a Node*, a string, and a pointer to a map owned by Huffman. Initially, these first two parameters will be a pointer to the root of the HuffTree and the empty string.

  This function will do a recursive PREFIX treewalk. At each step down the tree:

  1. If the left son of the Node* parameter is nullptr, you have reached leaf (a letter) and need to put its code into the codeMap. The string parameter is the code. Insert the pair (letter, code) into the map. Return.
  2. Otherwise, you have reached an interior node. Append a 0 to the string parameter and make a recursive call to walk the left son.
  3. Then append a 1 to the string parameter make a recursive call to walk the right son.

- 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 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.

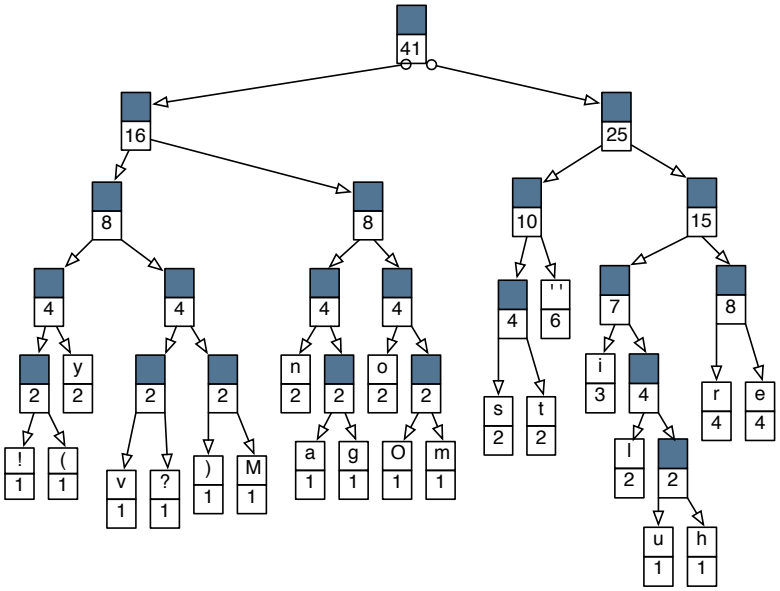

Figure 1: The Huffman tree, spread out neatly.

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		

Additions to the Huffman class. In P5, your `compress()` function created a tally array, moved the tallies into nodes, and organized the nodes into a heap. Add these actions after the HuffTree is made. Now append this to the code in the `compress() function`:

1. Call the heapToTree() function to convert the HuffHeap to a HuffTree.
2. Call the HuffTree::print() function to print the HuffTree in indented form.
3. Call the `mapCode()` function
4. Traverse the map and print the pairs, (letter, code-string).

3 Future Work

The last phase of the Huffman project is P7: Reopen the original text file. Encode each letter in it. Write the encodings to a binary file.