Ashley Nguyen - apn2my
PostLab10 - postlab10.pdf
4/20/16

Implementation

I created a huffmanNode class that would serve as the elements in my vector-based heap. I used the heap to construct the huffman tree. The reason I used a vector was mainly because it is an array based data structure. When implementing a tree where it is desirable to be able to find an element by tracing down the tree from the root, a vector provides a good mapping of where the elements are located in relation to their spots in the array. This rigid data structure lends to data compactness which is preferable over a dynamic memory implementation since we would not want the characters in our tree to shift around when attempting to decode and perhaps even encode. Since the fundamental principle of a Huffman tree relies on the most frequently occurring characters keeping towards the top of the tree, a vector implementation is relatively convenient for a structure that functions similarly to a cache. Vectors allow us to access the most recently used (or in Huffman encoding this would be the most frequently occurring) elements in order to properly encode its bit value by counting the steps down the tree to reach the element and the directionality of the traversal that would encode binary bits (either left or right path until reaching element). The tree data structure implemented based on the heap was chosen for my program for exactly this reason. The vector-based heap enables the structure to rigidly keep elements in an array and then construct them into a tree that would be able to heed the elements located closer to the root of the tree. The structures and reasons mentioned above culminate in implementing an encoding program and a decoding program. Although the decoding program was less complex than the encoding one, reusing the huffman Node class was vital in properly constructing another tree to recursively to map the encoded messages location on the tree.

Efficiency Analysis

Compression - Encoding

Worst case running time: To determine this I will evaluate each of the necessary functions in encoding a message; searching through an array of frequencies, insert into heap, removeMin() from heap, printPrefix() and setPrefix().

- An array of ints with size 128 held the values of the frequencies (int freq[128]) would yield a worst case run-time of **Big-Theta(n)** since to find a value would require a linear operation (step by step) through each of the elements in the array to get the appropriate frequency rate for the character.
- Since we read through the message to encode linearly, creating the tree requires multiple calls on insert() and removeMin() as the frequency rates for each letter may vary from the beginning of the message to the end of the message, inserting each distinct character and then updating its frequency as more of the message is read (possibly removing the current most frequent character once a MORE frequent character has been found). Thus both of these operations have a time complexity of **Big-Theta(log(n))**.
- The time complexity for printPrefix() would be **Big-Theta(n)** for my vector implementation since the function would print every single node/element in the vector linearly worst case.

- The time complexity for setPrefix() is also **Big-Theta(n)** linear since it needs to find the character to set (linear) and then assign the appropriate prefix for it.

Overall, based on the complexities for each of the functions used to create the Encoding tree the worst case time complexity if **Big-Theta(n^4(log(n))**.

Worst case space complexity: To find these values, I used the *sizeof* function in C++. Assuming worst case is a vector of size 100.
Node = 4 bytes (frequency) + 1 bytes (char) = 5 bytes + 8 bytes (pointers) = 13 bytes
Left & Right Pointers = 4 bytes + 4 bytes = 8 bytes
Frequency (Int) = 4 bytes
Letters (Char) = 1 bytes
Prefix (String) = Standard 4 bytes but depends on number of characters in the string so may be variable
Frequency Array of ints[128] = 4 bytes * 128 = 512 bytes
Heap = Node * size = 13 bytes * 100 = 1300 bytes
Total Space = 1842 bytes

Decompression - Decoding
Worst case running time: I will calculate this by looking at the complexity to read in a file, and create the Huffman Tree

- Reading in the file reads in each character until the end of the line is reached in a stepwise fashion thus this time complexity is linear, **Big-Theta(n).**
- Creating the Huffman Tree also linearly searches the encoded text file for a 0 or 1 bit and recursively creates the nodes based on these values. For example, to start, a 0 is read, then a node is recursively 'added' to the root's left pointer. Thus the time complexity for this operation is **Big-Theta(n)**. This is most likely possible due to the if statements in the createTree method that provide clear cut instructions for which cases to handle thus making this a relatively good time complexity worse case.

Overall, the worst case running time to decode is **Big-Theta(n)**.

Worst case space complexity:
Left-node = 13 bytes
Right-node = 13 bytes
Ascii array of characters [256] = 1 bytes * 256 = 256 bytes
Total Space = 282 bytes
Note that this is the worse case scenario, since decompression requires many memory manipulations to the tree, its memory should be dynamically allocated for proper decoding.